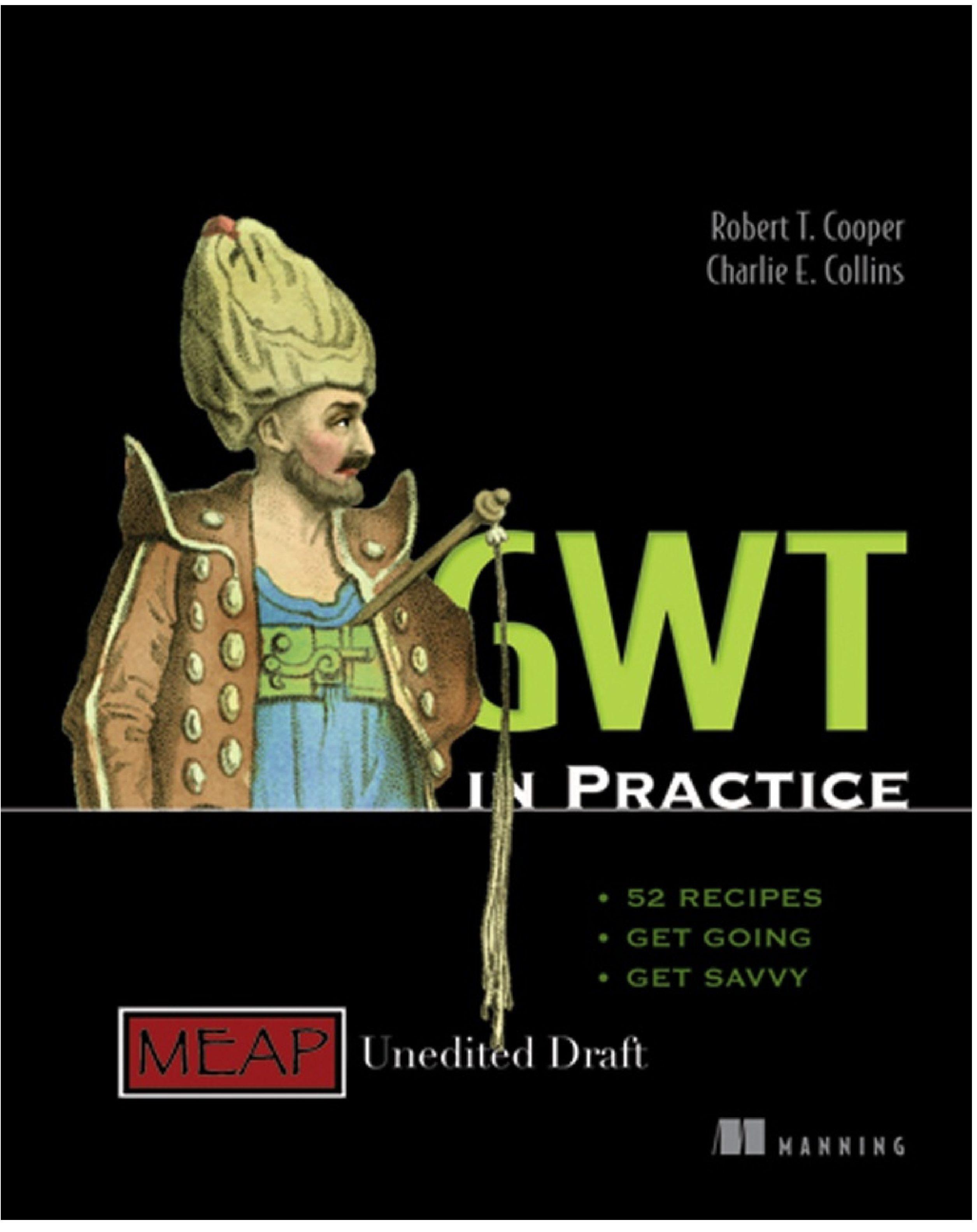
Robert T. Cooper
Charlie E. Collins
GWT
IN PRACTICE
• 52 RECIPES
• GET GOING
• GET SAVVY
MEAP
Unedited Draft
MANNING

MEAP Edition
Manning Early Access Program

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Google Web Toolkit In Practice

Table of Contents

Part1: Introductions

Chapter 1: Introducing GWT

Chapter 2: A New Kind of Client

Chapter 3: Communicating with the Server

Part2: Deeper Thoughts

Chapter 4: Core Application Structure

Chapter 5: Other Techniques for Talking to Servers

Chapter 6: Integrating Legacy and Third Party AJAX Libraries

Chapter 7: Building, Packaging, and Deployment

Chapter 8: Debugging, Testing, and Continuous Integration

Part3: Dirty Hands

Chapter 9: Java Enterprise Reinvented

Chapter 10: Building the Storefront

Chapter 11: Managing Application State

Introducing GWT

“The man of virtue makes the difficulty to be overcome his first business, and success only a subsequent consideration.”

Asynchronous JavaScript and XML (AJAX) development is hard. Not ascending Everest hard; maybe not even calculating your taxes hard, but it is hard. There are a number of reasons for this. JavaScript itself can require a lot of specialized knowledge and discipline. Browsers have slightly different implementations and feature sets. Tooling is still immature, and debugging in multiple environments is problematic. All of these factors add up to developers needing a vast knowledge of browser oddities and tricks to build and manage large AJAX projects.

To help deal with these problems, a number of toolkits and libraries have emerged. Libraries like Dojo, Script.aculo.us, Ext JS, and the Yahoo User Interface Library (YUI), have sought to provide enhanced core features and general ease of use to JavaScript. In addition, projects like Direct Web Remoting (DWR) have sought to simplify communications between the client and the server. And, even more advanced techniques, like those used by XML11 and Echo2, create an entire rendering layer in the browser while actually executing application code on the server side. These are all valid approaches, but the Google Web Toolkit (GWT) represents something different.

GWT is a Java to JavaScript cross compiler. That is, it takes Java code and compiles it into JavaScript to be run in a browser, as Figure 1.1 depicts.

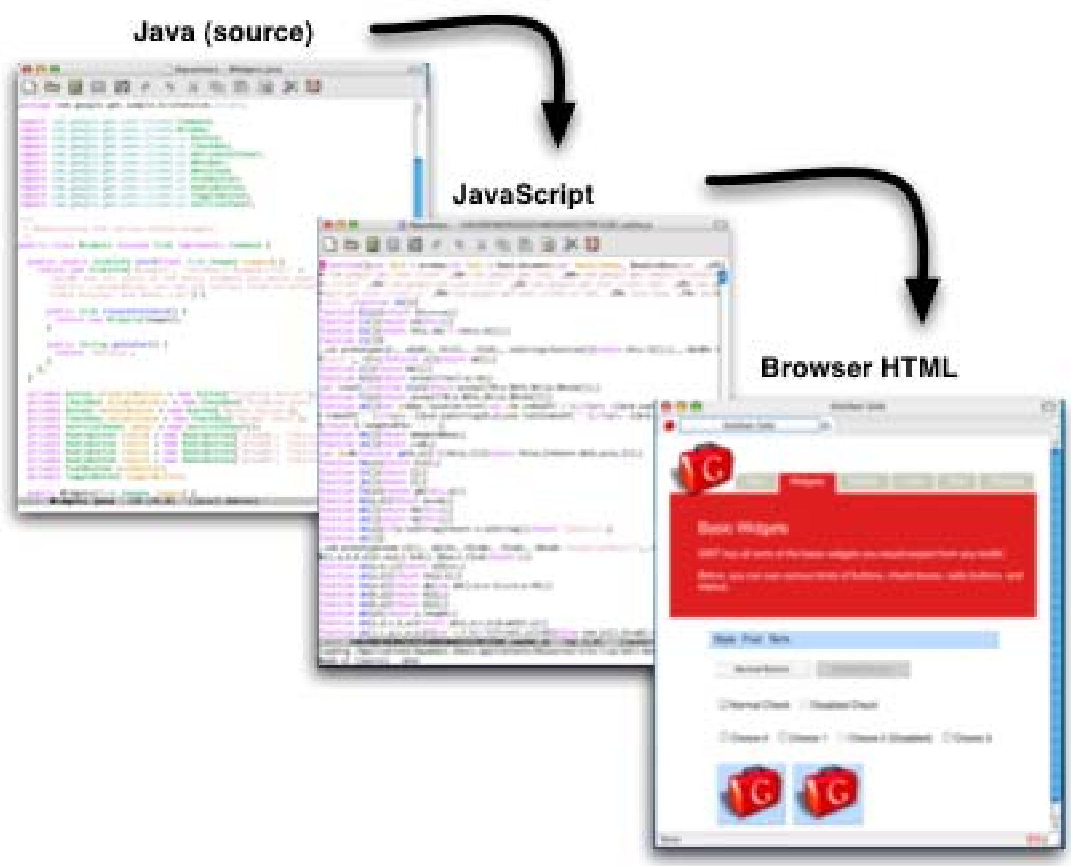

Figure 1.1 An overview of the GWT approach, Java source code that is compiled into JavaScript, which is then run in a web browser as JavaScript/HML/CSS.

There are many reasons GWT was engineered this way, starting from a statically compiled, strongly typed language like Java, with generous tooling and testing support, and emitting JavaScript application versions for all the major browsers in a compilation step. Chief among these reasons is the plain and simple fact that JavaScript is what is available in a browser, and starting from a single code base and generating all the required variations makes life a lot easier for the developer, and more consistent, stable, and performant for the user. But reigning in and leveraging JavaScript in the browser, starting from a Java code base, is not the only difference with GWT.

Other aspects that set GWT apart include: a harness for debugging Java bytecode directly as it executes in a simulated browser environment, a set of core User Interface (UI) and layout widgets with which to build applications, a Remote Procedure Call (RPC) system for handling communications with a host web server, internationalization support, and testing mechanisms.

GWT provides a platform to create true “Rich” Internet Applications (RIAs). Rich in the sense of allowing the client to maintain state and even perform calculations locally, with a full data model, without requiring a trip to the server for every update to the interface. This once again has many advantages for both the user and the developer. The user has a more responsive application, and the developer can distribute the load of the application. Also rich in terms of a wide variety of UI elements such as sliders, reflections, drag and drop support, suggest boxes, data bound tables, and more. This rich client platform,

ultimately utilizing only HTML, JavaScript, and Cascading Style Sheets (CSS), still has full access, in a variety of ways, to back end server resources when needed.

In short, GWT makes AJAX development a lot easier, not falling off a log easy; maybe not even taking candy from a baby easy, but easier - and it makes AJAX applications better for users. With *GWT in Practice*, we hope to take some of the facets of GWT that might seem hard or confusing and clarify them. Along the way we will also provide practical advice based on real world experience for many aspects of GWT development. If you are a Java web developer now, you are going to need something of a mind shift to build GWT applications well, and a good knowledge of the core GWT tools and how they work to make them work in your environment. If you are an AJAX developer coming to Java, you are going to need a bit of indoctrination in "The Java Way." Even if you are currently using or experimenting with GWT, you might want to increase your technical bag of tricks. It is our hope that you find these results, and maybe even a little more in this book.

1.1 Why GWT

The quick, but wrong, answer to why GWT was created and why it is gaining popularity, is because it's new and shiny! Though GWT is no "SOA" or "ESB" on the buzz meter yet, it is widely discussed. Is it deserving of the attention, and in many cases praise it receives? Getting past the hype, what are the reasons it was created, and why might it make sense to utilize it?

In the next few sections we will address these questions in order to lay out the overall approach of GWT, understand why it was created, where it is applicable, and why it should matter to you as a software developer (or manager). We will begin our discussion of what makes GWT significant with a brief trip back through the history web development, and the patterns and techniques involved, in order to frame the concepts.

1.1.1 History

In the beginning, there was HTML. Originally HTML was a semantic document markup intended to help the researchers on the Internet link between related documents. Soon after its usage blossomed, however, came forms. When forms were added to HTML, it transitioned from being strictly a document markup to being a user interface design language. HTML still suffers in some ways from this legacy, but the ease of deployment for web based applications became a driving factor in it's use nonetheless.

While the world of web applications began to expand, developers on the server side supporting the basic form applications created a new view of the Model View Controller (MVC) pattern centered on the server and rendered to HTML. From simple Perl scripts, to complete frameworks, development remained on the server, and things like application state became a complex problem, with complex solutions - like Struts to JSF and Seam. While these made web applications more capable, and development easier, they didn't provide the user experience of a full desktop-like "fat client" application, and continued in the same render-call-render lifecycle that the web had when forms were first added.

When Netscape introduced JavaScript to the web browser, it became something more than just a simple thin client. It became a platform on its own, capable of running small applications entirely within the scope of a page. This approach was widely used for simple things, such as field validation, but its use did not spread to more advanced functionality until Microsoft introduced the `XmlHttpRequest` (XHR) object. This made calls back to the server from the JavaScript environment easy, and the technique was

soon adopted by all the major browsers. The frontiers and capabilities of web applications then expanded even further when dynamic HTML and forms were combined with the server side, without requiring the entire browser window to be redrawn, via the usage of XHR. Thus the AJAX era was born.

1.1.2 Why AJAX Matters

AJAX changed the landscape because it finally broke the render-call-render pattern that existed previously and allowed browsers to update without making a visible, and often slow, trip back to the server for every page view. In fact, with AJAX even the term “page” is rendered a bit of a relic. AJAX moved the browser into a position much closer to being able to support full blown “fat” or “rich” Internet applications.

Because GWT is AJAX, and easily extensible to new browsers, it provides a wider array of supported devices than many other RIA approaches. Silverlight, Flash, and the Java Applet Plugin itself, all give developers the ability to create powerful, easy to use applications that run from a web browser. Nevertheless, their dependency on an environment outside of the browser means that they will always lag behind in deployments. GWT, and other AJAX based applications, worked on Day Zero for both the Nintendo Wii and the iPhone, for instance. GWT's concise browser abstraction also makes it easy to update as new versions of currently supported browsers are released, requiring only a recompile of the Java to support new devices.

AJAX is also significant in that applications that are native to the browser “feel” more natural to users. No matter how seamless the browser integration, typically there are some noticeable differences with plug in type technologies. Such differences may include: an install step; the treatment of browser constructs in a different manner, such as with bookmarks and navigation; and starkly different user interface elements - versus HTML, CSS, and JavaScript.

GWT, and AJAX in general, is a form of RIA that embraces the parts of the web that do work well, and that users are familiar with.

1.1.3 Leveraging the Web

Though the specifications that make up the web were adapted over the years to cope somewhat with drawbacks such as the stateless nature of HTTP, limited user input elements, and the render-call-render lifecycle, the advantages of the web still prevailed. Centralized management and delivery, no local installation, instant updates to both functionality and content, and concepts such as shareable “bookmarks,” are the reasons users, and developers alike, have embraced the web.

The desirable parts of the web, coupled with the implementation difficulties, are of course the reasons for the existence of the various RIA technologies in the first place, this includes GWT. Some of these technologies though, may, in some instances, go too far in their abstraction of the web layer – hiding the good, as well as the bad.

GWT, happily, is designed with the web tier in mind. The centrally hosted distribution model that the web offers is fully leveraged, and updates to applications are transparent and seamless in this model. The URL is the application download location, and the browser is the application platform. Concepts that users know and love are also present, and intentionally not hidden or disabled. Browser buttons, even the infamous back button, can be put to work rather than disabled – and do exactly what they are expected to do. Bookmarks are there, and again, they work – even when deep linked into a particular area of the

application.

Leveraging the parts of the web that do work well is intentional with GWT, as is the fact that GWT also extends the RIA landscape to provide tooling and testing support.

1.1.4 Tooling and Testing

Another of the reasons GWT is significant, and is different from some other RIA offerings, is that it provides tooling and testing support. GWT includes a powerful debugging shell that allows you to test and debug your code as it interacts with the native browser on your platform.

The testing support GWT provides is based on JUnit, and a few extensions the toolkit provides. Your GWT code can be tested as Java, from the shell. After you compile your code into JavaScript, the same test can also be used again in that form, using further scaffolding provided by GWT. This allows you to test on various browser versions, and if so desired, even different platform and browser combinations.

The use of the Java language also comes with first-class tooling support. Invaluable tools such as code parsers like PMD, static analysis tools like Checkstyle and FindBugs, advanced refactoring engines available in most Java Integrated Development Environments (IDEs), and debuggers and profilers, all function perfectly normally within the context of the GWT shell.

Tooling support and testing facilities, which are front and center with GWT, are pretty standard fare in traditional programming, but are not as common in terms of client side web technologies. Along with this support, GWT provides a great deal of help for developers in other areas as well. One of the biggest advantages GWT offers is that it helps you cope with browser differences.

1.1.5 A Single Code Base

Traditionally, web development has required a doctorate in Browserology in order to be aware of, and cope with, all of the differences in behavior among the different browser types and versions. Along with knowing details about multiple versions of the HTML, Document Object Model (DOM), CSS, JavaScript, and HTTP standards, developers have also always needed to be aware of the way quirks and bugs affect each browser. And, all of that knowledge was required just to write a standard web application using HTML and forms.

Add to the mix XML and XHR, and more browser differences, and you can easily see why AJAX development has a well deserved reputation “on the street” - difficult. GWT doesn't avoid that difficulty directly, but it does encapsulate it and allow developers to worry about their application, rather than the differences among the browsers. GWT lets you work on a single code base, using Java, and then cross-compile that code into AJAX enabled HTML and JavaScript for most of the major browser types and versions currently in use (Internet Explorer, FireFox, Safari, and Opera at present).

This is a huge benefit to developers. Imagine a world where browser makers actually adhere to standards, and use the latest one consistently – where you, as a developer, need to only create one version of your application and it works regardless of the user agent. Well, that is a fantasy world of course, but the next best thing is to write application code once, and then generate additional code to cope with the differences in the browser environment. This gives you, in effect, the same thing – one version of code to write instead of multiple. This is what GWT aims for.

The abstraction is not always perfect, but it does work well a large percentage of the time. And, when there is a corner case type problem, a particular browser version having an issue in a particular scenario, it

can usually be resolved quickly with open access to the source, and the expertise of the community (and then that expertise is put back into the toolkit so that others avoid the same problem). The potential issues with the browser difference abstraction bring us to another general topic concerning GWT, some additional limitations, both real and perceived.

1.1.6 Limitations

Along with the potentially leaky, but undeniably extremely useful, abstraction of the web layer, GWT has a few other potential limitations or drawbacks you should also be aware of.

Don't call GWT a framework (even though in places the documentation does). It hasn't been here for years, and a lot of what people expect from “Framework” products simply aren't there. Java web frameworks, like Struts and WebWork, evolved from the lessons learned in traditional Java web development. As such, they come with a style, and a set of patterns and expected ways of working. This clearly defined approach helps developers see where to begin with their application. GWT doesn't have that. When working with GWT as a developer, it is best to think of it as an analogue to the Java Abstract Windowing Toolkit (AWT). It is the basis for your application framework, not the framework itself.

This can be thought of as a limitation, but it can also be seen as a well focused starting point. GWT does not try to dictate the development approach you will use, rather it simply aims to provide the components for you to create AJAX applications, or frameworks, on your own. Through the course of this book we will demonstrate several patterns for development that should mitigate concerns in this area, but the real point here is that a toolkit is not necessarily bad, it's a starting point.

One area that is a limitation with GWT is that search engines have a hard time with JavaScript. Most search engine agents/robots do not speak JavaScript, and or know how to navigate an application not composed of simple links. This means many GWT resources are simply non-existent as far as such search engines are concerned. Of course, this applies to all of AJAX and JavaScript in general, not just GWT.

There are many techniques to cope with this. Often the most useful and thorough is simply to have a native HTML-only version of your application, alongside your GWT application, so that user agents that cannot load JavaScript, whether or not they are search engine agents, can still access your data. The non-JavaScript version of your application need not be “pretty,” but rather should concentrate on core content and any degree of functionality you can provide. Be careful though; make sure any non-JavaScript version of your site is an accurate representation. If you provide “search engine only” content, this is known as “cloaking,” and it can get your site banned from search engine indexes.

When viewed in total the benefits of GWT generally outweigh the limitations, in many, but not all cases. Overall GWT provides a new approach to building AJAX web applications - starting from a single code base, generating the variations for all the major browsers, embracing the concepts of the web that are already familiar to users, and providing many tools and supporting features along the way. Getting beyond the inspiration and reasoning we will next turn to addressing exactly what it is that GWT includes.

1.2 What GWT Includes

The Google Web Toolkit provides a number of technologies for building AJAX applications. We are going to look briefly at each of them now, and then we will step through them and hit the highlights of their usage. These items include the GWTCompiler, a UI layer, an RPC system, several additional

utilities that make the web tier easier to manage, and the GWTShell. To begin, at the core of GWT is the compiler.

1.2.1 GWTCompiler

The GWT Java compiler takes Java code and compiles it into JavaScript. That is really it. It has some advanced rules for doing this, however. By defining GWT compile tasks into "modules," which we will cover in more detail in Section 1.3, the compiler can perform more analysis on the code as it is processed, and branch into multiple compilation artifacts for different output targets. What does this mean? When compiling a class you can specify differing implementations based on known parameters. The obvious switch point is the user agent, or client browser you are targeting. This feature drives the core of GWT's cross-browser compatibility.

Unlike most AJAXs toolkits written in JavaScript, which have complex logic to adapt to different browser environments, GWT will switch out the implementation of core classes based on the user agent for which the compilation task is building. Though we will learn more about the compiler in general throughout the book, including a bit more detail later in this chapter in Section 1.5, the bottom line is that each browser gets a lean, mean, and specific version of your application and isn't forced to download code for all the other browsers your application can support. It is through this mechanism that GWT implements a cross-browser UI toolkit.

1.2.2 User Interface Layer

Built on top of this intelligent compilation system is a cross-browser user interface layer. The real magic here comes from implementing the UI elements in Java, then using a browser-specific implementation of the core DOM to build out the native browser elements as they are needed by the higher level Java layer. While some AJAX libraries have a lot of focus on UI widgets, GWT is intended to provide a core of UI functionality which the users and community can build upon.

The GWT UI layer provides a wide variety of layout related panels, data representation constructs such as `Tree` and `Grid`, a set of user input elements, and more. The 1.4 release of GWT began to expand the UI toolkit to include some new advanced elements, like a rich text editor, and a suggest box. This release also started to include some great new optimized UI elements that draw from the power of the plug-in capable compiler, such as the `ImageBundle`.

The `ImageBundle` takes static images in your application and merges them into a single graphic file. Then by using the placed background mode in the CSS box model, shows only the part of the large single image required at any point, as shown in Figure 1.2. This means the client browser can make a single request to get all the images in your application, rather than negotiating multiple HTTP request-response cycles, thereby improving the startup time of your application.

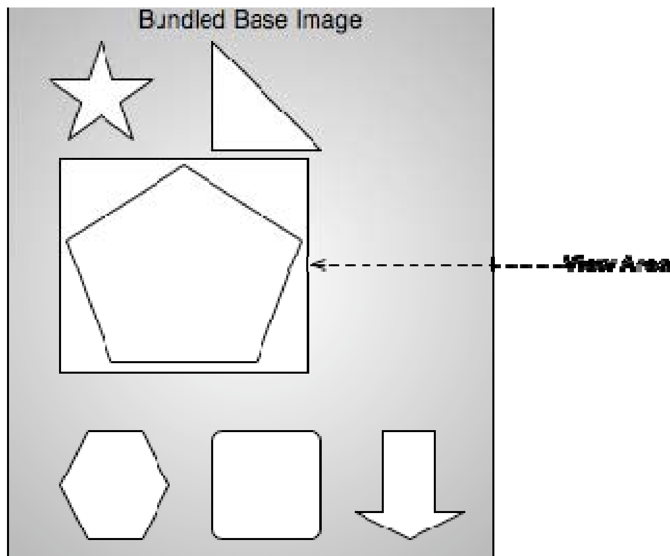

Figure 1.2 The BundledImage packages many images into one and then renders on the page by positioning the image as a background behind a view area transparent image.

In addition to the the core UI foundation, and the subset of UI elements provided, GWT also includes several means for communicating with server resources. Chief among these methods is the GWT Remote Procedure Call (RPC) mechanism.

1.2.3 Remote Procedure Calls

Another GWT core feature that draws heavily from the plug-in capabilities of the compiler is the RPC functionality. This system allows for serialization and deserialization of Java objects from server-side implementations of remote services, which can then be called asynchronously from the client.

Here the compiler generates code during the compilation step to handle the serialization at a low level. Serialized objects are versioned and mapped at compile time. This carries with it two major advantages. First, you can guarantee the client and server agreement as new versions are deployed. Secondly, this allows the server implementation to compress the state of Java objects down to arrays of JavaScript primitives. This passing of simple arrays allows for even more concise data formatting than JavaScript Object Notation (JSON), which many laud for its simplicity and efficiency. We will take a close look at the RPC functionality, and communicating with servers, in Chapter 3.

Along with it's own RPC system, GWT also includes a set of additional utilities that make development for the web simpler.

1.2.4 Additional Utilities

Beyond the approach and the core elements, GWT also includes a number of utilities that are designed to make building applications for the web tier a bit easier. These include support for additional ways to communicate with servers, internationalization (I18N) tools, a History management system, and testing support.

GWT provides several client side libraries for going beyond the standard GWT RPC, and instead

communicating with XML and JSON based services. These make it easy to integrate with many existing web APIs, and or allow you to implement a completely non Java based back end. We will explore these techniques in more detail in Chapter 5.

There is also a compile-time checked internationalization library that makes providing multi-language support easy and reliable. This makes it not only possible, but very straightforward, to internationalize your AJAX web applications. We will see a little bit more about how this system works later in this chapter in Section 1.5. when we delve into compiler details.

The History management system makes bookmarking and deep-linking in your AJAX application pretty easy too. This system can also be overloaded to use when you need to seamlessly link between several deployed GWT applications. This can come in handy if you ever find the need to “lazy load” portions of your application for performance or initial download time reasons (though these occasions really should be quite rare, based on all the other optimizations GWT provides).

Last but not least, there is formal testing support. GWT gives you a means to test your code, by writing test cases, either as Java, or as JavaScript. We will cover this in more detail in Chapter 8, which is devoted to testing, and other aspects of code quality such as continuous integration.

All of the additional utilities GWT provides are aimed at making the development cycle a little easier, and a little more predictable, on the web tier. Along these same lines, one of the main “tools” GWT provides is the GWTShell.

1.2.5 GWTShell

All of the great features GWT includes build on the core of the architectural approach and the GWT compiler. “But wait! There's more!” GWT also includes a nice set of developer tools starting with the GWT hosted mode shell, which is shown in the screen shot in Figure 1.3.

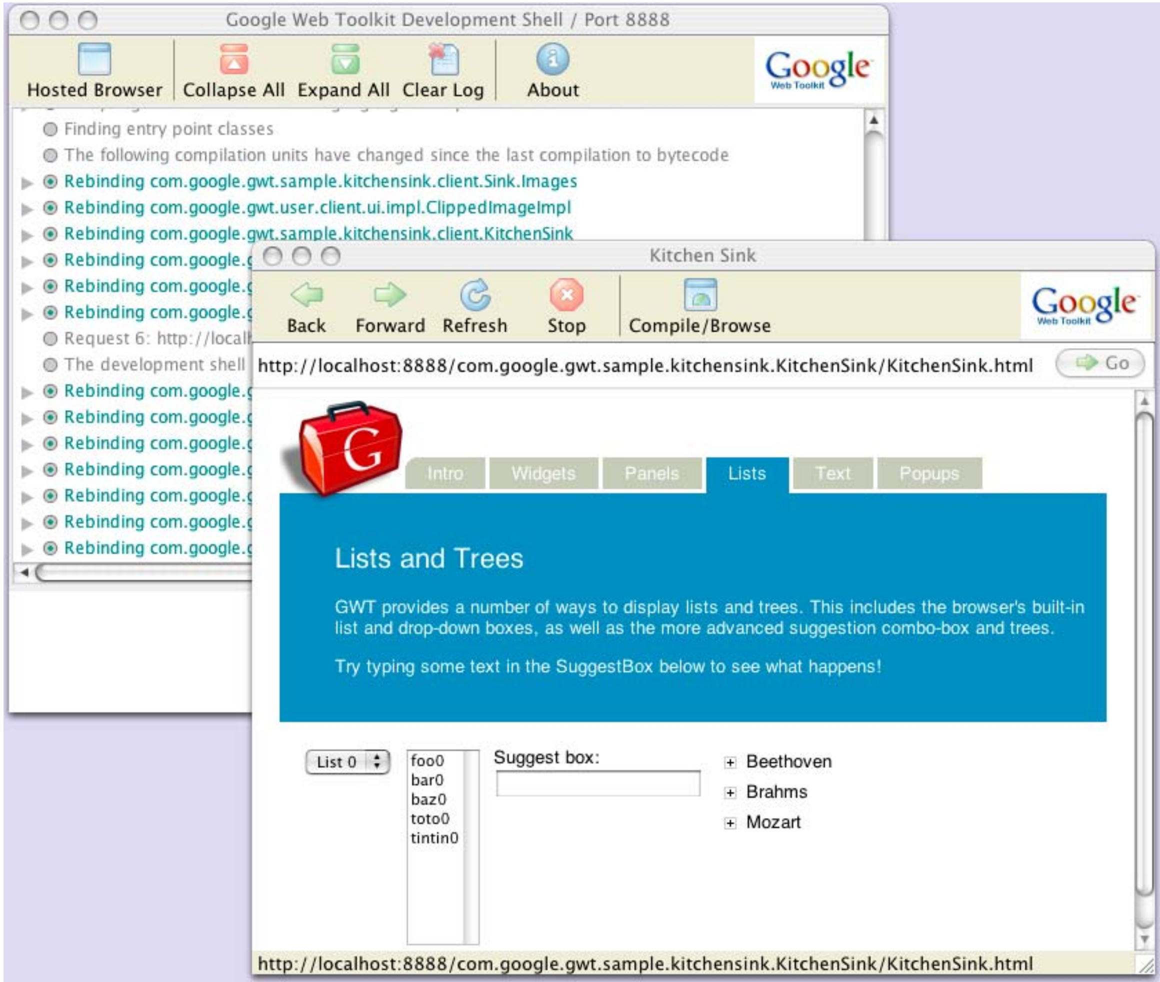

Figure 1.3 Screenshot of the GWTShell and hosted mode browser. The shell includes a hierarchical log view and a custom web browser.

The GWTShell allows you to test your application in a browser while executing the native Java bytecode. This gives you the ability to use all your favorite Java tools to inspect your application, including profilers, step-through debugging, and JTI-based monitors. This hosted mode browser, with an embedded Apache Tomcat server, is also what makes possible the ability to test your compiled JavaScript with JUnit. Because it is so central to all GWT development projects, we will cover the shell in more detail later in this chapter in Section 1.4, and we will make use of it throughout the book.

Since we have seen what GWT entails, in terms of the reasons for the architecture choices in general, and in terms of some of the tools and utilities provided, we will now turn to the basic concepts surrounding working with GWT and getting a project started.

1.3 GWT Basics

Individual GWT projects are composed of a few key parts. Here we will briefly touch on the main components of a GWT project, to get you familiarized with the concepts that you will use over and over again when working with the toolkit.

First, GWT projects are defined in terms of bundled resources and configuration into packages known as “modules.” The configuration modules provide is used both when projects are compiled, and when they are run. Beyond configuration, modules also make possible a rich inheritance mechanism. Because of this capability, projects can be complete web “applications,” or also can be of a pure library nature, or anywhere in between.

Part of what modules define are the starting point for a project's code, known as an entry point. Entry point classes are coded in Java, and end up referenced by a module and compiled to JavaScript. Modules themselves, and the entry points they define, are invoked through a `<script>` reference on an HTML page, known as a host page. Host pages invoke GWT projects and also support a few special meta tags that can be used to tweak things. At a high level, these are the three main components of a GWT project: a module configuration file, an entry point class, and an HTML host page.

In the next few sections we will introduce each of these components, beginning with modules, and the inheritance they enable.

1.3.1 Modules and Inheritance

There are several aspects to GWT that may seem alien to new users. The main one is the use of the “module” concept to define GWT applications. One of the core problems with web development over the years has been getting true reusability with applications and components. The GWT modules system allows application components to package the client side application code, server side service implementations, and assets such as graphics and CSS files, into an easily re-distributable package.

Modules are also an important part of the GWT bootstrap and deployment process, including their facilitation of the key concept of inheritance. Modules are defined by an XML file placed along with the Java source implementing the module. This XML file declares several primary elements: inherited modules, servlet deployments, compiler plug-ins, and entry points. Listing 1.1 shows a sample module file.

Listing 1.1 Calculator.gwt.xml module definition

```
<module>

    <inherits name='com.google.gwt.user.User'/>                      #1
    <entry-point                                                     #2
           class='com.manning.gwtip.calculator.client.Calculator'/>
    <stylesheet src='com.manning.gwtip.calculator.style.css'/>       #3

</module>
```

(annotation) <#1 Inherit the core GWT classes>
(annotation) <#2 Define the EntryPoint>
(annotation) <#3 Specify a CSS file to use>

This module file is pretty basic. It comes from the example application we will create in Chapter 2, a GWT based calculator. The `<inherits>` tag tells GWT to inherit the core “User” classes **#1**. This includes the GWT UI elements, as well as the custom compiler elements that generate the appropriate versions for Firefox, Safari, Internet Explorer, and Opera. Remember, inheriting a module brings in all the elements of the module, not just source code. This is one of the main reasons for having the module system. Of course the compiler can find additional Java sources referenced by your project, but the module system allows build time and server side behavior to be inherited as well. Next is the declaration

of an `<entry-point>` **#2**. As mentioned previously, an entry point is the starting point for code for a GWT project – we will cover this in more detail in the next section. Finally, is the inclusion of a style sheet for the application **#3**. While it is not necessary to provide style information at this level, a style sheet declared in the module follows the module as it is inherited, making common formatting easier to carry from application to application as common components are reused.

This module definition makes a few core assumptions, the main one being the directory layout of the source folders. While customizable, the default and conventional layout for GWT applications follows the following format:

package folder
- `Module.gwt.xml` : The module definition file.
  - `client` : The client package containing code that will be cross compiled to JavaScript
  - `public`: A folder containing assets (images, CSS files, etc) used by the module
  - `server`: The package containing server-side implementation code
  - `rebind`: The package containing compile-time code (Generators, etc)

In the above example, referencing "`com.manning.gwtip.calculator.style.css`" means the " `com.manning.gwtip.calculator.style.css` file in the `public` folder of the module package." You can also see that the `<entry-point>` definition references a class in the `client` package.

Though we have really only scratched the surface with regard to modules here, don't worry, beyond this introduction we will come back to them later in this chapter when we discuss the GWT compiler, in the next chapter when we build our first sample application, and throughout the book as we complete further examples. Looking past modules for now, in the next section we will look more closely at entry points, and the GWT bootstrap process involving host pages.

1.3.2 HostPages

To run your GWT application, you start with an HTML page. The HTML "host page," as it is known in GWT parlance, contains just a couple of special elements needed to activate a GWT application. First, is a reference to a "nocache" JavaScript file that GWT generates for every module. This is a JavaScript file that detects the appropriate version of the application module based upon user agent, and other factors, and loads up the monolithically compiled application. Listing 1.2 shows an example HTML host page, again taken from the calculator example we will build in the next chapter.

Listing 1.2 The Calculator HTML Host Page

```
<html>
      <head>
            <title>Wrapper HTML for Calculator</title>
      </head>
      <body>
            <script language="javascript"
              src="com.manning.gwtip.calculator.Calculator.nocache.js"> #1
            </script>
            <iframe id="__gwt_historyFrame"                            #2
                 style="width:0;height:0;border:0"></iframe>
            <h1>Calculator</h1>
```

```
        </body>
</html>
```

(annotation) <#1 The bootstrap JavaScript file>
(annotation) <#2 A special invisible iframe for history>

The important part is that the page includes the “bootstrap” JavaScript file that starts up the application. **#1** When you compile your GWT application you will get two versions of this file. They are named [ModuleName].nocache.js, and [ModuleName]-xs.nocache.js, respectively. These two files typically represent the only part of your GWT application that you should send to the client without HTTP level caching enabled. All of the other files GWT generates are named based on a hash of the source code that went into creating them, meaning they can be cached indefinitely by the client with no ill effects. The two “nocache” files determine which hashed version needs to be loaded when the page is requested.

The “xs” version of the nocache file is for use by other domains that might include your application. This file allows communication to your server side resources, even if the HTML page that includes the script was not served from the same host as the server side resources. This short circuiting of the “Same Origin Policy” is dangerous though. It opens up your application to possible Cross Site Scripting (XSS) security problems. It also makes your application more “Mashupable” (which may, or may not, be desirable). In short, you should really only use this script if you know what your are doing. We will discuss other aspects of the Same Origin Policy, and ways to work around it, in Chapter 6.

When a module nocache script (either version, standard or “xs”) is loaded from a host page, the remainder of the GWT bootstrap process is invoked and loads an `EntryPoint` class.

1.3.3 Entry Point Classes

An entry point is a simple client-side class that implements the `com.google.gwt.core.client.EntryPoint` interface, which defines a single method: `onModuleLoad()`. This is the GWT equivalent of a class with “`public static void main()`” declared in Java – a place to begin. Again borrowing from the calculator example we will build in Chapter 2, Listing 1.3 shows an example `EntryPoint` implementation.

Listing 1.3 The entry point for the Calculator class

```
public class Calculator implements EntryPoint {
    public void onModuleLoad() {
        RootPanel.get().add(new CalculatorWidget("calculator"));   #1
    }
}
```

(annotation) <#1 Add a CalcualtorWidget to the <body> >

While the `onModuleLoad()` method itself is pretty simple, the operative element here is the use of GWT's RootPanel class. This is your application's access to the HTML host page. Calling `RootPanel.get()` returns a default container inserted just before the `</body>` tag in the HTML page **#1**. If you want finer-grained control over where a GWT widget is injected into the page, you can call `RootPanel.get("ElementID")` where “ElementID” is a String value denoting the ID of an element on the HTML page, typically a `<div>` tag.

Modules, host pages, and entry points are the fundamental high level concepts involved with any

GWT project. We have touched on these concepts here, to cover the basics, but don't worry if this introduction seems brief. We will being using these components, and expanding on the, in the next chapter when we build our first GWT example application, and throughout the book in subsequent examples. With our initial tour of these elements now complete, next we will take a closer look at the the tool you will use the most during development: The GWTShell.

1.4 Working With the GWTShell

The GWTShell, which we have already briefly introduced in Section 1.2.5, is one of the most important components GWT provides, and one you will use every day when developing GWT applications. Because of this, we will cover it in more detail here, and we will touch on it again in other chapters when specific situations warrant it.

The shell is composed of three main parts: a logging console, an embedded Tomcat server, and the hosted mode browser. The GWTShell console provides an enhanced logging interface and centralized GUI as a GWT dashboard. The “hosted mode” browser is capable of invoking your Java classes directly on browser events, rather than requiring a compilation to JavaScript, which lets you use a standard Java debugger to work with your AJAX code, instead of relying solely on compiled JavaScript for testing and interaction. The development server, Tomcat “Lite facilitates local development and testing of server based resources, (this is covered in detail in Chapter 3).

GWTShell supports several common command line parameters you should be familiar with, these parameters are shown in the help output below:

```
Google Web Toolkit 1.4.60
GWTShell [-port port-number] [-noserver] [-whitelist whitelist-string]
[-blacklist blacklist-string] [-logLevel level] [-gen dir] [-out dir] [-style
style] [url]

where
  -port       Runs an embedded Tomcat instance on the specified port (defaults
to 8888)
  -noserver   Prevents the embedded Tomcat server from running, even if a port
is specified
  -whitelist  Allows the user to browse URLS that match the specified regexes
(comma or space separated)
  -blacklist  Prevents the user browsing URLS that match the specified regexes
(comma or space separated)
  -logLevel   The level of logging detail: ERROR, WARN, INFO, TRACE, DEBUG,
SPAM, or ALL
  -gen        The directory into which generated files will be written for
review
  -out        The directory to write output files into (defaults to current)
  -style      Script output style: OBF[USCATED], PRETTY, or DETAILED (defaults
to OBF)
and
  url         Automatically launches the specified URL
```

We will use many of these options in the examples throughout this book, as we encounter each option

we will go into more detail. The first of these parameters you may want to tweak, and the first we will explore, `-logLevel`, changes the output level of the logging console, which is the first thing you notice when GWTShell starts up.

1.4.1 The Logging Console

The GWTShell console is a hierarchical logging display with a few simple buttons that invoke the hosted mode browser and control logging output. The logging display is controlled by the `-logLevel` option on the GWTShell command line. Valid log levels are shown in table 1.1, the default level is INFO.

Table 1.1 GWTShell -logLevel level options

Log Level	Description
ERROR	Shows only critical errors in the GWTShell code.
WARN	Shows uncaught exceptions in user code. Warn and Error information are displayed in red in the shell window.
INFO	Shows server startup information and invocations into specific GWT modules. Most of the time what you will see in this mode is simply "Starting HTTP on port 8888".
TRACE	Shows each request logged, as well as module instantiations, their locations on the class path, and the time. This is perhaps the most useful mode for day to day development. Trace and Info level information are displayed in gray in the shell window.
DEBUG	Shows binding of classes inside the GWTShell for code invocations and URL mapping. This mode is also useful for debugging compilation errors into JavaScript, should you encounter them. Debug information is displayed in green in the shell window.
SPAM	Shows all `ClassLoader` events and invocations to native JavaScript. Spam information is displayed in teal in the shell window.
ALL	Shows all.

Figure 1.4 is a sample screen shot of the GWTShell console component showing the buttons and some sample logging output.

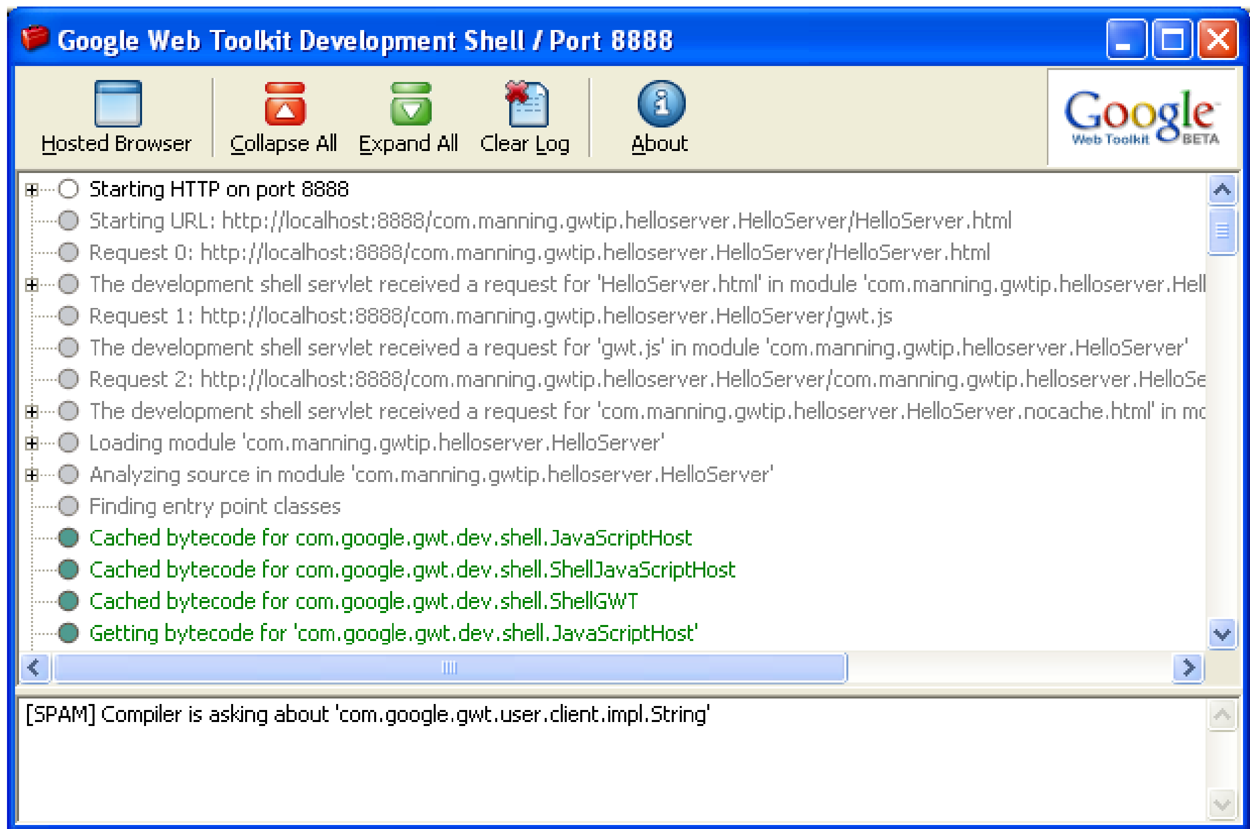

Figure 1.4 The GWTShell Logging Console shows different log level messages in varying colors.

From the GWTShell console GUI, you also have the option to invoke the hosted mode browser. The hosted mode browser is what allows you to explore your Java application using a browser and browser based events.

1.4.2 The Hosted Mode Browser

The hosted mode browser component operates as a test browser harness that directly invokes Java binary code in response to browser events. This allows you to short-circuit the compilation to JavaScript step and immediately see changes to your code, as well as perform step through debugging of code between the client and the server sides.

The hosted mode browser also provides a shortcut to executing the compiled JavaScript version of your application; this is known as “web mode.” While you are using the hosted browser from GWT, you can click the “Compile/Browse” button to perform a complete compilation of your Java to JavaScript, and browse your hosted application in the shell's development web server.

Note

It is important to make sure you have the GWT_EXTERNAL_BROWSER environment variable set before you select “Compile/Browse”. For example (on Linux):

```
export GWT_EXTERNAL_BROWSER=/usr/bin/firefox
```

This specifies the command line GWTShell will invoke to launch a browser against the hosted Tomcat.

You can also use the hosted mode browser apart from the GWTShell's Tomcat instance with the `-noserver` option. This tells the shell not to use the embedded Tomcat instance in hosted mode, but rather to use an external server that you specify. Using an external web server, or Java container, can have several advantages. You can, for example, use the standalone browser to debug and test Java user interface code that is talking to PHP or .NET back-ends that can't otherwise be run in GWTShell. Or you can use a Java back-end that may run in a different container than Tomcat. Using an external server with `-noserver` is covered further in Chapter 3, when we get into more detail concerning configuring the embedded Tomcat instance in general.

It is recommended, for hosted mode use, regardless of which container you use, that you name your context path the same as your module, for example "`com.manning.gwtip.helloserver.HelloServer`." Doing this will make it easier to map your service servlet calls later.

Along with the shell, and the hosted mode browser that it includes, the next key GWT tool/concept that we need to fill in with a bit more detail is the GWTCompiler.

1.5 Understanding the GWTCompiler

The GWT compiler really is the fulcrum of GWT. The entire tack GWT takes, encapsulating browser differences and compiling JavaScript from Java, is all made possible by the design and architecture of the compiler. Because of the importance of the GWTCompiler, and it's central role in virtually every aspect of the GWT approach, we need to elaborate on the compiler further here – and we will also come back to it and fill in other details as we encounter them in additional examples throughout the book.

Though the GWT compiler compiles Java into JavaScript, it's important to understand that GWTCompiler doesn't compile Java the same way as `javac`. The GWT compiler is really a **Java source to JavaScript source translator**.

The GWT compiler, partly because it operates from source, needs hints about the work that it must perform. These hints come in the form of the module descriptor, the marker interfaces that denote serializable types, the JavaDoc style annotations used in serializable types for Collections, and more. Although these hints may sometimes seem like overkill, they are needed because the GWT compiler will optimize your application at compile time. This doesn't just mean compressing the JavaScript naming to shortest possible form, but also includes pruning unused classes, and even methods and attributes, from your code. The core engineering goal of the GWT compiler is summarized succinctly:

> You pay for what you use.

This has big advantages over other AJAX/JavaScript libraries, where a large initial download of a library might be needed for just a few elements that might be used. In Java, serialization marked by the `java.io.Serializable` interface is handled at the bytecode level. GWT examines your code and only provides serialization for the classes where you will explicitly need it.

Like the GWTShell, GWTCompiler supports a set of useful command line options which are as follows:

```
Google Web Toolkit 1.4.60
```

```
GWTCompiler [-logLevel level] [-gen dir] [-out dir] [-treeLogger] [-style
style] module

where
  -logLevel    The level of logging detail: ERROR, WARN, INFO, TRACE, DEBUG,
SPAM, or ALL
  -gen         The directory into which generated files will be written for
review
  -out         The directory to write output files into (defaults to current)
  -treeLogger  Logs output in a graphical tree view
  -style       Script output style: OBF[USCATED], PRETTY, or DETAILED
(defaults to OBF)
and
  module       Specifies the name of the module to compile
```

The `-gen` and `-out` command line options specify where generated files and the final output directory are to be, respectively. And `-logLevel`, again like GWTShell, is used to indicate the level of logging performed during the compilation. You can even use the `-treeLogger` option to bring up a window to view the hierarchical logging information like you would see contained in the shell's console display.

The GWTCompiler supports several styles of output, each of use in looking at how your code is executing in the browser.

1.5.1 JavaScript Output Style

When working with the GWTCompiler, there are several values you can use with the `-style` command line option to control what the generated JavaScript looks like. These options are as follows:

`OBF` – For Obfuscated. This is a non-human readable, compressed, version suitable for final deployment.
`PRETTY` – Pretty-printed JavaScript with meaningful names.
`DETAILED` – Pretty-printed JavaScript with fully qualified names indicated.

To give you an idea of what these mean, let's take a look at examples of `java.lang.StringBuffer`, compiled in the three different modes. First, in Listing 1.4, is the OBFuscated mode.

Listing 1.4 StringBuffer in OBFuscated compilation

```
function A0(){this.B0();return this.js[0];}
function C0(){if(this.js.length > 1){this.js = [this.js.join('')];this.length
= this.js[0].length;}}
function D0(E0){this.js = [E0];this.length = E0.length;}
function Ez(F0,a1){return F0.yx(yZ(a1));}
function yB(b1){c1(b1);return b1;}
function c1(d1){d1.e1('');}
function zB(){}
_ = zB.prototype = new f();_.yx = w0;_.vB = A0;_.B0 = C0;_.e1 = D0;_.i =
'java.lang.StringBuffer';_.j = 75;function f1(){f1 = a;g1 = new iX();h1 = new
iX();return window;}
```

Obfuscated mode is just that. This is intended to be the final compiled version of your application, which has names compressed and whitespace cleaned. Next is the PRETTY mode, shown in Listing 1.5.

Listing 1.5 StringBuffer in PRETTY compilation

```
function _append2(_toAppend){   #1
  var _last = this.js.length - 1;
  var _lastLength = this.js[_last].length;
  if (this.length > _lastLength * _lastLength) {
    this.js[_last] = this.js[_last] + _toAppend;
  }
   else {
    this.js.push(_toAppend);
  }
  this.length += _toAppend.length;
  return this;
}

function _toString0(){   #2
  this._normalize();
  return this.js[0];
}

// Some stuff omitted.

function _$StringBuffer(_this$static){
  _$assign(_this$static);
  return _this$static;
}

function _$assign(_this$static){
  _this$static._assign0('');
}

function _StringBuffer(){
}

_ = _StringBuffer.prototype = new _Object();
_._append = _append2;
_._toString = _toString0;
_._normalize = _normalize0;
_._assign0 = _assign;
_._typeName = 'java.lang.StringBuffer';             #3
_._typeId = 75;
```

(annotation) <#1 append() becomes _append2() to avoid collision>
(annotation) <#2 toString() becomes _toString0()>
(annotation) <#3 _typeName holds the name of the original Java class>

Pretty mode is useful for debugging as you can see method names are somewhat preserved. However, collisions are resolved with suffixes, as the `_toString0()` method above shows **#2**. Lastly we have the

final style mode the GWTCompiler provides, DETAILED, this is displayed in Listing 1.6.

Listing 1.6 StringBuffer in DETAILED compilation

```
function java_lang_StringBuffer_append__Ljava_lang
            _String_2(toAppend){  #1
  var last = this.js.length - 1;
  var lastLength = this.js[last].length;
  if (this.length > lastLength * lastLength) {
    this.js[last] = this.js[last] + toAppend;
  }
   else {
    this.js.push(toAppend);
  }
  this.length += toAppend.length;
  return this;
}

function java_lang_StringBuffer_toString__(){  #2
  this.normalize__();
  return this.js[0];
}

function java_lang_StringBuffer_normalize__(){
  if (this.js.length > 1) {
    this.js = [this.js.join('')];
    this.length = this.js[0].length;
  }
}

// . . . some stuff omitted

function java_lang_StringBuffer(){
}

_ = java_lang_StringBuffer.prototype = new java_lang_Object();
_.append__Ljava_lang_String_2 =
java_lang_StringBuffer_append__Ljava_lang_String_2;
_.toString__ = java_lang_StringBuffer_toString__;
_.normalize__ = java_lang_StringBuffer_normalize__;
_.assign__Ljava_lang_String_2 =
java_lang_StringBuffer_assign__Ljava_lang_String_2;
_.java_lang_Object_typeName = 'java.lang.StringBuffer';
_.java_lang_Object_typeId = 75;
```

(annotation) <#1 Line broken for length>
(annotation) <#2 Method names are fully qualified>

Detailed mode preserves the full class name, as well as the method name **#2**. For overloaded methods, the signature of the method is encoded into the name, as the `.append()` method above **#1**.

There are some important concepts to grasp about this compilation structure, especially in light of what will be covered in later sections of this chapter concerning the way GWT interacts with native

JavaScript, through the JavaScript Native Interface (JSNI). Naming of your classes and methods in their JavaScript form is not guaranteed, even between different compilations of the same application. While use of the special syntax provided with JSNI will allow you to invoke known JavaScript objects from your Java code, and invoke your compiled Java classes from within JavaScript, you cannot freely invoke your JavaScript when using obfuscated style, predictably. This does impose certain limitations on your development:

1. If you intend to expose your JavaScript API for external use, you need to create the references for calls into GWT code using JavaScript Native Interface registrations. You will see how to do this later in the book.
2. You cannot rely on JavaScript naming in an object hash to give you `java.lang.reflect.*` type functionality since the naming of methods will not be reliable.
3. While a rarity, you should bear in mind - especially if you are publishing using the `PRETTY` setting - potential conflicts with other JavaScript libraries you are including in your page.

In addition to the compiler output options that are available and how they affect your application, you should also be aware of a few other nuances associated with the compiler.

1.5.2 Additional Compiler Nuances

Currently, the compiler is limited to J2SE 1.4 syntactical structures. This means that exposing generics or annotations in your GWT projects can cause many problems. For many of the things you might wish to use annotations for, there are other options. You can often use JavaDoc style annotations, to which GWT provides its own extensions. Along with the J2SE 1.4 limitations, you also need to keep in mind the limited subset of Java classes that are supported in the GWT Java Runtime emulation (JRE) library. This library is growing, and there are third party extensions, but you need to be aware of the constructs you can use in client side GWT code – the complete JRE you are accustomed to, is not available.

Of course, one of the great advantages of GWT's tack of compiled JavaScript from plain Java is that you get to leverage your existing toolbox while building your AJAX application. While executing the hosted mode browser, you are running regularly compiled Java classes. This, again, means you can use all the standard Java tooling – static analysis tools, debuggers, IDEs, and the like.

While these tools are a good idea for any code you are writing, they become even more important in a GWT world. This is because cleaning up your code means less transfer time to high latency clients. It is also worth mentioning that JavaScript is a fairly slow execution environment, so such cleanup can have a large impact on ultimate performance. The GWTCompiler helps in this capacity by optimizing the JavaScript it emits to include only classes and methods that are on the execution stack of your module, and by using browser native functions where possible, but you should still always keep the nature of JavaScript in mind. To that end, we will now take a closer look at the lifecycle of a GWT compilation and how this JavaScript is generated.

1.5.3 The Compiler Lifecycle

When the GWT compiler runs, it goes through several stages for building the final compiled project. These are where the need for the GWT module definition file becomes very clear. First, the compiler identifies what combinations of files will need to be built. Then, it generates any client side code using the Generator metaprogramming model. Lastly, it produces the final output (as seen in Listings 1.4-1.6, our previous output style examples). Next, we will take a look at each of these steps in the compiler life-cycle in more detail.

Identify Build Combinations

One of the great strengths of GWT is that it builds very specific versions of the application, each targeted to exactly what the client needs (user agent, locale, so on). This keeps the final download, and the operational overhead at the client level, very lean. A particular build combination is defined in the GWT module definition using a `<define-property>` tag. This establishes a base set of values that begin being used for the build. The first and very obvious one is the user agent that the JavaScript will be built for. Listing 1.7 shows the core GWT `UserAgent` module definition.

Listing 1.7 The GWT UserAgent definition

```
<define-property name="user.agent"
        values="ie6,gecko,gecko1_8,safari,opera"/>                              #1

  <property-provider name="user.agent"><![CDATA[                                #2
      var ua = navigator.userAgent.toLowerCase();
      var makeVersion = function(result) {
          return (parseInt(result[1]) * 1000) + parseInt(result[2]);
      };

      if (ua.indexOf("opera") != -1) {                                          #3
        return "opera";
      } else if (ua.indexOf("webkit") != -1) {                                  #4
        return "safari";
      } else if (ua.indexOf("msie") != -1) {                                    #5
        var result = /msie ([0-9]+)\.([0-9]+)/.exec(ua);
        if (result && result.length == 3) {
          if (makeVersion(result) >= 6000) {
            return "ie6";
          }
        }
      } else if (ua.indexOf("gecko") != -1) {                                   #6
        var result = /rv:([0-9]+)\.([0-9]+)/.exec(ua);
        if (result && result.length == 3) {
          if (makeVersion(result) >= 1008)
            return "gecko1_8";                                                  #7
          }
        return "gecko";
      }
      return "unknown";
  ]]></property-provider>
```

(annotation) <#1 Define the valid options>

(annotation) <#2 Establish property-provider JavaScript implementation>
(annotation) <#3 Detect for Opera>
(annotation) <#4 Detect for Safari>
(annotation) <#5 Detect for MSIE>
(annotation) <#6 Detect for Gecko>
(annotation) <#7 Detect for Gecko 1.8>

Here the `<define-property>` tag establishes a number of different builds that the final compiler will output **#1**. In this case, `ie6`, `gecko`, `gecko1_8`, `safari`, and `opera`. This means each of these will be processed as a "build" of the final JavaScript that GWT emits. GWT can then switch implementations of classes based on properties using the `<replace-with>` tag in the module definition. GWT determines which of these will be used as the GWT application starts up using the JavaScript snippet contained within the `<property-provider>` tag **#2**. This snippet will be built into the startup script that determines which compiled artifact is loaded by the client.

At the core of the GWT UI classes is the `DOM` class. This gets replaced based on the `user.agent` property. Listing 1.8 shows this definition.

Listing 1.8 Changing the DOM implementation by User Agent

```
<inherits name="com.google.gwt.user.UserAgent"/>

<replace-with class="com.google.gwt.user.client.impl.DOMImplOpera">         #1
      <when-type-is class="com.google.gwt.user.client.impl.DOMImpl"/>
      <when-property-is name="user.agent" value="opera"/>
</replace-with>

<replace-with class="com.google.gwt.user.client.impl.DOMImplSafari">        #2
      <when-type-is class="com.google.gwt.user.client.impl.DOMImpl"/>
      <when-property-is name="user.agent" value="safari"/>
</replace-with>

<replace-with class="com.google.gwt.user.client.impl.DOMImplIE6">           #3
      <when-type-is class="com.google.gwt.user.client.impl.DOMImpl"/>
      <when-property-is name="user.agent" value="ie6"/>
</replace-with>

<replace-with class="com.google.gwt.user.client.impl.DOMImplMozilla">       #4
      <when-type-is class="com.google.gwt.user.client.impl.DOMImpl"/>
      <when-property-is name="user.agent" value="gecko1_8"/>
</replace-with>

<replace-with class="com.google.gwt.user.client.impl.DOMImplMozillaOld">    #5
      <when-type-is class="com.google.gwt.user.client.impl.DOMImpl"/>
      <when-property-is name="user.agent" value="gecko"/>
</replace-with>
```

(annotation) <#1 DOM implementation for Opera>
(annotation) <#2 DOM implementation for Safari>
(annotation) <#3 DOM implementation for MSIE>
(annotation) <#4 DOM implementation for Gecko 1.8>
(annotation) <#5 DOM implementation for Gecko>

Now you can see the usefulness of this system. The basic `DOM` class is implemented with the same interface for each of the browsers, providing a simple core set of operations upon which cross-platform

code can be easily written. Classes replaced in this method cannot be used using simple constructors, but must be created using the `GWT.create()` method. In practice, the `DOM` object is a singleton exposing static methods that are called by applications, so this still is invisible. This is an important point to remember if you want to provide alternate implementations based on compile time settings in your application. You can also define your own properties and property providers for switching implementations. We have found that doing this for different runtime settings can be useful. For example, defining a "debug," "test," and "production" setting, and replacing some functionality in the application based on this property can help smooth development in certain cases.

This technique of identifying build combinations, and then spinning off into each specific implementation, during the compile process, is known in GWT terms as "deferred binding." The GWT documentation sums this approach up as: "Deferred Binding is the Google Web Toolkit answer to Java reflection." Dynamic loading of classes (dynamic binding) is not truly available in a JavaScript environment, so GWT provides another way. For example, `obj.getClass().getName()` is not available, but `GWT.getTypeName(obj)` is. The same is true for `Class.forName("MyClass")`, which has a counterpart of `GWT.create(MyClass)`. Using deferred binding, the GWT compiler can figure out every possible variation, or "axis," for every type and feature needed at compile time. Then, at runtime the correct permutation for the context in use can be downloaded and run.

Remember, though, each axis you add becomes a combinatory compile. If you use 4 languages, and 4 browsers versions, you must now compile 16 final versions of the application (and if you use several "runtime" settings you add that many more combinations to the mix. This concept is depicted in Figure 1.5.

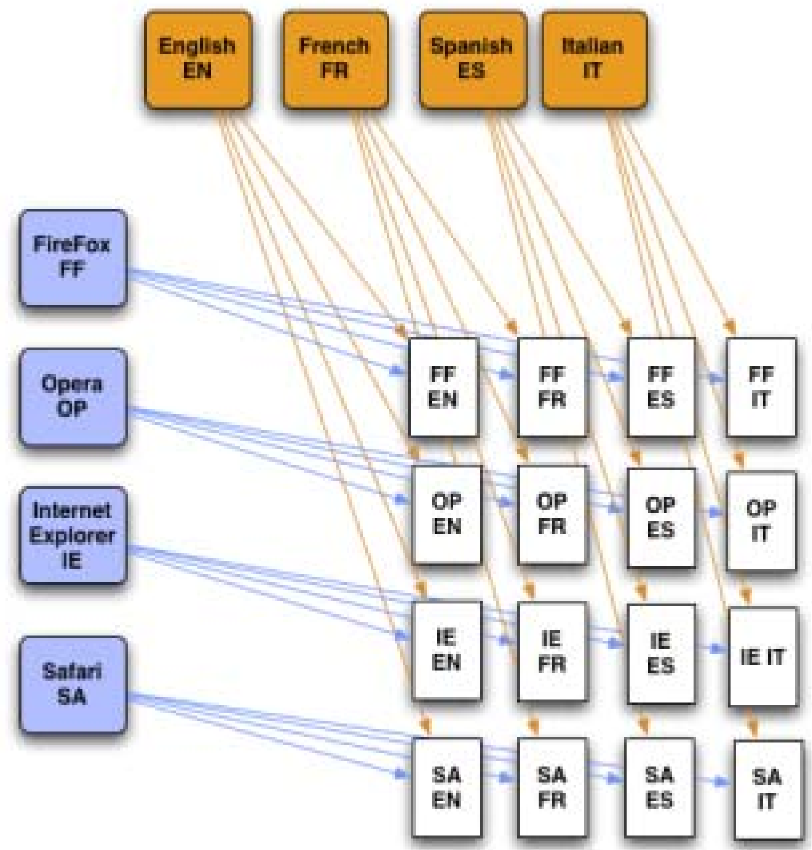

Figure 1.5 Multiple versions of a GWT application are created by the compiler for each "axis" or variant application property, such as user agent, and locale.

Compiling a new monolithic version of your application for each axis involved, doesn't affect your end users negatively at all. Rather, this technique allows each user to download only the exact version of your application they need, without forcing any unused portion to come along for the ride. Though this is beneficial for users, it does slow compile time considerably, and can be a pain point for developers.

Reducing the Compile Variants to Speed Up Compile Time

Even though GWT compile time can be long, keep in mind that the end result for users is well optimized. And, the GWT module system allows you to tweak the compile time variants for the situation. During day to day development you may want to use the <set-property> tag in your module definition to confine things to a single language, or single browser version, to speed up the compile step.

Another important use for module properties is in code generation, which is the next step of the compilation process.

Generate Code

GWT's compiler includes a code generation/metaprogramming facility that allows you to generate code based on module properties at compile time. Perhaps the best example of this is the internationalization support. The I18N module defines several no-method interfaces that you extend to

define Constants, Messages (which include in-text replacement), or Dictionaries (which extract values from the HTML host page). The implementations of each of these classes are built at compile time using the code generation facility, producing a lean, custom version of your application in each language. The I18N module does this through the <extend-property> tag which lets you add additional iterative values to a property in a module. Listing 1.9 demonstrates using this concept to add French and Italian support to a GWT application.

Listing 1.9 Defining French and Italian using extend-property

```
<extend-property name="locale" values="fr" />
<extend-property name="locale" values="it" />
```

When an application inherits the I18N module, the GWT compiler searches for interfaces that extend one of the I18N classes, and generates an implementation for the class based on a resource bundle matching the language code. This is accomplished via the <generate-with> tag in the I18N module's definition. Listing 1.10 shows this, and the <property-provider> tag, which is used for establishing which language will be needed at runtime.

Listing 1.10 The I18N Module's locale Property Declarations

```
<define-property name="locale" values="default" />                  #1

<property-provider name="locale">                                   #2
      <![CDATA[
      try {
      var locale;

      // Look for the locale as a url argument
      if (locale == null) {
        var args = location.search;
        var startLang = args.indexOf("locale");
        if (startLang >= 0) {
          var language = args.substring(startLang);
          var begin = language.indexOf("=") + 1;
          var end = language.indexOf("&");
          if (end == -1) {
            end = language.length;
          }
          locale = language.substring(begin, end);
        }
      }

      if (locale == null) {
        // Look for the locale on the web page
        locale = __gwt_getMetaProperty("locale")
      }

      if (locale == null) {
        return "default";
      }

       while (!__gwt_isKnownPropertyValue("locale",  locale)) {
```

```
          var lastIndex = locale.lastIndexOf("_");
          if (lastIndex == -1) {
          locale = "default";
            break;
          } else {
            locale = locale.substring(0,lastIndex);
          }
        }
        return locale;
      } catch(e){
        alert("Unexpected exception in locale detection, using default: " + e);
        return "default";
      }
        ]]>
</property-provider>

<generate-with class="com.google.gwt.i18n.rebind.LocalizableGenerator">     #3
      <when-type-assignable class="com.google.gwt.i18n.client.Localizable" />
</generate-with>
```

(annotation) <#1 Define the locale property>
(annotation) <#2 Establish the property-provider>
(annotation) <#3 Define the generator for Localizable>

The module first establishes the "locale" property **#1**. This is the property we extended in Listing 1.9 to include "it" and "fr." Next, the property provider is defined **#2**. The value is checked first as a parameter on the request URL in the format "locale=xx," then as a meta tag on the host page in the format `<meta name="gwt:property" content="locale=x_Y">`, and finally it defaults to "default."

The last step is to define a generator class. Here it tells the compiler to generate implementations of all classes that extend or implement `Localizable` with the `LocalizableGenerator` **#3**. This class writes out Java files that implement the appropriate interfaces for each of the user's defined `Constants`, `Messages`, or `Dictionary` classes.

Notice that to this point, nowhere have we dealt specifically with JavaScript, outside of small snippets in the modules. It is in the final step that GWT will produce the JavaScript.

Produce Output

While it should be clear from the above subsections that a great deal can be done simply from Java with the GWT module system, at some point, if you wish to integrate existing JavaScript, or extend or add lower level components, you may need to get down to JavaScript level implementations. For this, GWT includes JSNI. This is a special syntax that allows you to define method implementations on a Java class using JavaScript. Listing 1.11 shows a simple JSNI method.

Listing 1.11 A simple JSNI Method

```
      public class Alert {

          public Alert() {
              super();
          }
          public native void alert(String message)                    #1
```

```
        /*-{                                                    #2
            alert(message);
        }-*/;
    }
```

(annotation) <#1 Define using the native keyword>
(annotation) <#2 Surround with special comment>

Here we first define the `alert()` method using the “native” keyword. This, much like JNI in regular Java indicates that we are going to use a native implementation of this method. Unlike JNI, the implementation lives right in the Java file, surrounded by a special /*- -*/ comment syntax. To reference Java code from JavaScript implementations, syntax very much like JNI is provided. Figure 1.6 shows the structure of calls back into Java from JavaScript.

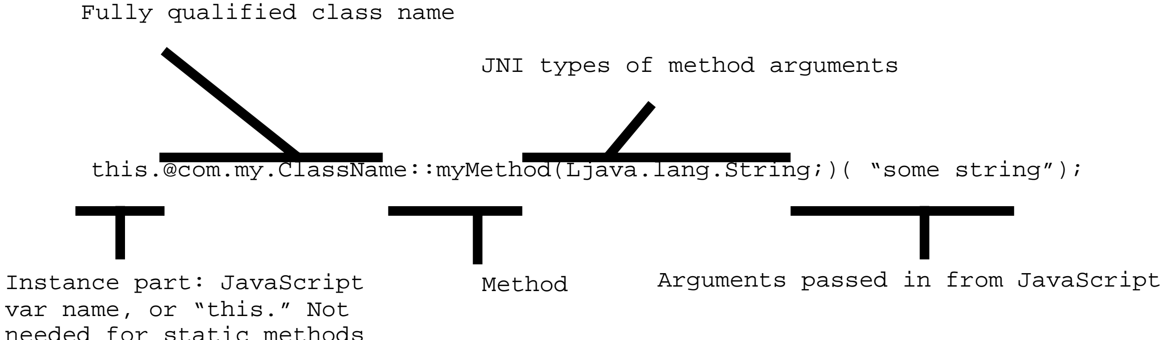

Figure 1.6 The structure of JavaScript Native Interface call syntax.

GWT reuses the JNI typing system to reference Java types. We will look into JSNI in more detail in Chapter 6. As you saw above, the final compiled output will have synthetic names for methods and classes, so the use of this syntax is important to make sure that GWT knows how to direct a call from JavaScript.

In the final step of the compilation process, GWT takes all the Java files, provided and generated, and the JSNI method implementations, and examines the call tree, pruning unused methods and attributes as discussed above. Then it transforms all of this into a number of unique pay-for-what-you-use JavaScript

files, targeted very specifically at each needed axis. This minimizes both code download and execution in the client. While this complex compilation process can be time consuming for the developer, it assures that the end user experience is the best it can possibly be for the application that is being built.

Though we will come back to certain compiler aspects as we work through concrete examples later in the book, our discussion of the compiler life-cycle, deferred binding, generators, compiler output and JSNI, and some additional nuances, completes our initial introduction of the core of GWT.

1.6 Summary

GWT is much more than another AJAX library or another web development tool. GWT adopts a fundamentally different approach by moving familiar development patterns around, providing a cross compiler to JavaScript, and making greater usage of the web browser as a "chubby client." This represents a step between the "fat client" three-tier architecture of the 1980s and the "thin client" architecture of both the 1970s and 1990s.

GWT borrows from the approaches that have come before it and takes things in a new direction, expanding once again the web development frontiers. All the while, GWT maintains the advantages of traditional compiled language development by starting out from Java, and adopts the successful component oriented development approach – applying these concepts to the web tier in a responsive AJAX fashion.

Along with starting from Java, GWT also embraces the parts of the web that have worked well and allows developers and users to remain on familiar ground. This is one of the most overlooked, yet very significant aspects of GWT. GWT does not try to hide the web from you, just to achieve the moniker "rich" web application. Instead GWT happily integrates with and uses HTML, JavaScript, and CSS.

Enhancing the tooling for developing the rich web applications Google is known for certainly makes sense, but why share these tools with the world? This reasoning is not complicated, but it is impressive and refreshing nonetheless. Brett Taylor, the GWT product manager, sums it up quite nicely as, "what's good for the web is good for Google." This sort of "rising tide" approach makes sense for a web company such as Google.

In the next several chapters that compose Part 1 of this book, we will break down the components of GWT, which have been introduced in this chapter. Chapter 2 gets you up and running with a basic example of a client-side GWT application, which reinforces the core concepts we have introduced here, and covers important design patterns concerning utilizing the web browser in a new way - as a fully featured client. Then, in Chapter 3, we cover GWT Remote Procedure Calls, object serialization, and learn to communicate with servers. In Chapter 4 we cover a small but complete application, to reinforce the front-to-back canonical GWT approach.

Then, in Part 2 of the book, we delve more into practical details. There we discuss building, packaging, and deploying GWT applications, and we look deeper at many of the GWT tools and features. These include other means for communicating with server, more involved JSNI, and testing and continuous integration. Finally, in Part 3 of the book, we present several complete example applications, which further cover both client and server side elements, including data binding, advanced UI concepts, streaming, and integration with traditional JEE components.