15 Changing applications based on GWT properties

This chapter covers

- Managing browser-specific code
- Internationalization (dynamic and static)
- Altering application functionality based on locale
- Implementing and using user defined properties

So far in your journey, you've used GWT properties without making too much out of them. GWT properties are defined in various module XML files and are used at compile time to change your application based on their values. The most obvious case of this in action is GWT's ability to work with many browsers. GWT provides a `user.agent` property that defines, in the `com.google.gwt.user.UserAgent` module XML file, a number of values for different browsers. At compile-time, the compiler produces a JavaScript permutation for each value of the `user.agent` property value, replacing a number of core class files with browser specific versions. These replacements are defined in various other module XML files—for example, `com.google.gwt.user.DOM` defines which browser-specific DOM class is used.

NOTE GWT properties are designed to be used at compile-time to help produce various permutations of your application as well as at load-time to help select the appropriate permutation. They aren't intended to be accessed via your code at runtime (if you're trying to do the latter, then you should think of redesigning your application).

You can also use GWT properties in your programming. Internationalization is one area where you've seen this already: You extended the property values of the `locale` property to include the Swedish locale for the Dashboard application. In this chapter, we'll look at how you can use GWT properties to your advantage, including creating a compact Flash widget (which sends only the necessary code to the browser). We'll explain fully how you can change parts of the application based on the locale and look at the full range of internationalization. Finally, we'll introduce our own GWT property to indicate whether the Dashboard application is executing in an intranet or Internet mode (the mode indicated alters the functionality of the application).

Before we zoom into the chapter, let's recap the information about properties from chapter 9 to save you having to flip back through the book.

15.1 Quick summary of properties

Properties are defined in the module XML file. You can set up an initial set of values initially using the `define-property` tag, such as this one:

```
<define-property name="property-name" values="val1,val2,..."/>
```

It's possible to extend this set in later modules that inherit the module where the properties are defined, using the `extend-property` tag:

```
<extend-property name="property-name" values="additional-values,..."/>
```

Once the property values are defined, you need to identify which of those values you should use; you can do that either declaratively or programmatically. Declaratively, you set property values in the application's HTML file through an HTML `meta` tag:

```
<meta name='gwt:property' content='property-name=value'>
```

You can also provide some simple JavaScript code in the module XML file that returns valid values from the set of property values based on a calculation. For example, you can define a `property-provider` tag as follows that returns a `property-value` based on the success of a particular condition statement:

```
<property-provider name="property-name">
<![CDATA[
   if (condition1 == true) return value1
   if (condition2 == true) return value2
   // other code
   else return valuen
]]>
</property-provider>
```

The programmatic approach is used to decide which browser is in use and then set the appropriate value for `user.agent`.

Now that we've recapped properties, it's time to see how you can put them to use as opposed to letting GWT have all the fun. We start our discussion by looking at how you can plug into the way GWT manages browser differences by building a Flash widget.

15.2 Managing browser differences

Throughout this book, we've referred to the benefit of GWT as a write-once, run-across-many-browsers type of affair. This is true, and don't be misled by the title of this section—we aren't saying you should go out of your way to write applications that behave differently in different browsers. But on some occasions, you need to manage the fact that differences exist between the way browsers perform actions (most often at the DOM or JavaScript level).

We don't think you'll need to deal with browser differences often, unless you're implementing specific functionality that requires it. You'll do it in this section, for example, to build a widget that displays a Flash movie. In normal web pages, you need to send code for both Internet Explorer and other browsers to ensure that the movie is displayed; using GWT, you can send only the code needed. Before we move on to the widget, let's look at how GWT manages browser differences to see what you can reuse.

15.2.1 How GWT manages browser differences

We'll look at the DOM implementation and see how GWT deals with browser differences, and we'll examine the patterns you can use if you need to. The hierarchy of classes provided for the GWT DOM implementation is shown in figure 15.1.

You can see a number of classes in this hierarchy, some marked with specific browser names—Safari, Opera, and so on, as well as Standard and plain Impl. All these different classes provide browser-specific implementations of DOM manipulation methods. GWT uses our old friend deferred binding in order to create an instance of the DOM classes it uses:

```
DOMImpl impl = (DOMImpl) GWT.create(DOMImpl.class);
```

This allows the compiler to select the appropriate class for the browser as directed by the DOM module XML definition. The module XML file contains a number of replace tags (see chapter 9 for a complete definition) like the following:

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

The `replace` statement tells the compiler to replace the `DOMImpl` class with the `DOMImplIE6` class if the `user.agent` property is set to a value of `ie6`.

In this section, you'll build a widget that performs in a similar manner, sending the necessary code to display a Flash movie in Internet Explorer or all the other browsers.

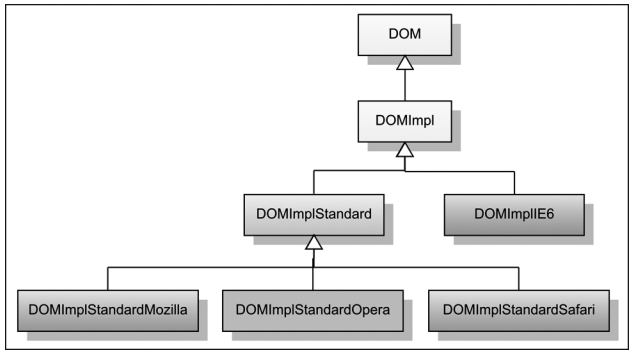

Figure 15.1
Class hierarchy of DOM classes in the GWT distribution

15.2.2 Building the Flash widget

When displaying Flash movies in a web browser, you must use two different tags to ensure that cross-browser functionality is achieved. Internet Explorer needs to use the `OBJECT` tag, and the other browsers use the `EMBED` tag. Figure 15.2 shows the simple Flash movie used in the Dashboard—we specifically set the text to distinguish the browser (*IE6* for Internet Explorer and *Other* for Firefox, Opera, and Safari).

Normally, you write both tags in the HTML page, and the browser ignores the irrelevant one, as shown in listing 15.1.

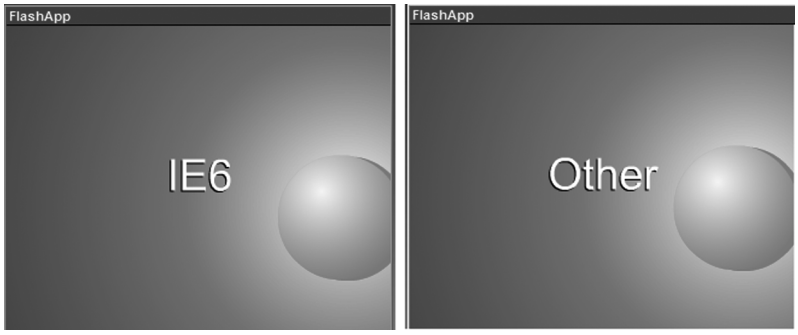

Figure 15.2 Two versions of our test Flash movie, one showing the word *IE6* and the other the word *Other*. You'll use GWT's browser-specific code capabilities to show the right movie in the right browser.

Listing 15.1 Standard way of dealing with browser differences when loading a Flash movie

```
<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/
          swflash.cab
        #version=6,0,40,0"
        WIDTH="550" HEIGHT="400" id="myMovieName">
   <PARAM NAME=movie VALUE="myFlashMovie.swf">
   <PARAM NAME=quality VALUE=high>
   <PARAM NAME=bgcolor VALUE=#FFFFFF>
   <EMBED src="/support/flash/ts/documents/myFlashMovie.swf"
          quality=high bgcolor=#FFFFFF
          WIDTH="550" HEIGHT="400"
          NAME="myMovieName" ALIGN=""
          TYPE="application/x-shockwave-flash"
          PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer">
   </EMBED>
</OBJECT>
```

But you can do better than that by using GWT browser-specific code: You can create a widget whose implementation is set up for either Internet Explorer or for all the other browsers. First, you'll create a simple Flash movie using one of the free online Flash generators (we chose this approach because we're not exactly the world's best artists!). The benefit of the movie is that it allows you to change the text as a parameter to the movie's URL, and thus you can change the movie text as shown in the two images in figure 15.2 so you can easily see what browser you're running in.

You'll follow the pattern used in the majority of GWT browser-specific code, in that you create a main class that instantiates one of the specific implementation classes to provide the functionality. The appropriate implementation is chosen by the compiler based on `replace` tags you'll place in the component application's module XML file.

Creating the Flash widget

The Flash widget is a simple extension of the `Composite` class that uses deferred binding to get a reference to a `FlashMovieImpl` class, the implementation class, as shown in listing 15.2. You set up the code so it can be made flexible in the future with the ability to pass in a set of parameters in the `FlashMovieParameters` class, but for now this isn't used.

Listing 15.2 Creating a simple Flash widget

```
public class FlashMovie extends Composite{

   FlashMovieImpl impl =
      (FlashMovieImpl) GWT.create(FlashMovieImpl.class);

   public static class FlashMovieParameters{
      public String movieName;
   }

   public FlashMovie(FlashMovieParameters params){
      SimplePanel panel = new SimplePanel();
      String tag = impl.createMovie(params);
      DOM.setInnerHTML(panel.getElement(), tag);
      initWidget(panel);
   }

   public FlashMovie(){
      new RuntimeException("Need Flash Movie Parameters!");
   }
}
```

1 Access implementation class

2 Implement constructor

Prevent constructor with no parameters

In the widget, you create an instance of the implementation class ❶, which is then used in the constructor to create the appropriate tag for the browser ❷. In the constructor, you create a simple panel in which the Flash movie will reside, and then you obtain the appropriate tag from the implementation. Once you have that, you use DOM manipulation to place the tag in the HTML or the panel and then initialize the panel as the widget.

Next, we'll look at the two implementation classes: one for the majority of browsers and the other specifically for IE.

Implementing the standard Flash widget

The standard implementation for the Flash widget creates the `EMBED` tag, as shown in listing 15.3, by stringing together a number of `String`s. In a real implementation, you would use the parameters passed in to set dimensions, and so on, but, in this example, you hard-code all the parameters. Note that in the URL, you set the title value to be the word *Other*, which is what you expect to be displayed on screen.

Listing 15.3 Defining the Flash widget's implementation class for the majority of browsers supported by GWT

```
public class FlashMovieImpl {
   public String createMovie(FlashMovieParameters params) {
      String theMovie = "";
      theMovie += "<EMBED src=\"flash_movie.swf?slogan=&
                          slogan_white=&title=Other&
                          title_white=Other&url=\" ";
      theMovie += "width=\"318\" height=\"252\" play=\"true\"
                   loop=\"false\"";
      theMovie += " quality=\"high\" ";
      theMovie +=
    "pluginspage=\"http://www.macromedia.com/go/getflashplayer\">";
      theMovie += "</EMBED>";
      return theMovie;
   }
}
```

Now that you've defined the standard implementation, you need to provide the IE-specific implementation.

Implementing the Internet Explorer–specific Flash widget

The IE implementation returns the `OBJECT` tag equivalent of the standard implementation. The two key things to note with the code shown in listing 15.4 is that it

must extend the standard implementation and that its URL sets it up to display the text *IE6*.

Listing 15.4 Defining the Flash widget's implementation class for Internet Explorer

```
public class FlashMovieImplIE6 extends FlashMovieImpl{
   public String createMovie(FlashMovieParameters params) {
      String movie =
            "<OBJECT classid=
                     \"clsid:D27CDB6E-AE6D-11cf-96B8-444553540000\"
                     codebase=\"http://download.macromedia.com/pub/
                     shockwave/cabs/flash/swflash.cab#\\
                     version=6,0,40,0\" name=\"demoMovie\"
                     id=\"demoMovie\" width=\"318\" height=\"252\">";
      movie += "<param name=\"movie\" value=flash_movie.swf?slogan=&
                       slogan_white=&title=IE6&title_white=IE6&
                       url= />";
      movie += "<PARAM name=\"play\" value=\"true\" />";
      movie += "<PARAM name=\"loop\" value=\"false\" />";
      movie += "<PARAM name=\"quality\" value=\"high\" />";
      movie += "</OBJECT>";
      return movie;
   }
}
```

With the implementation files in place, it's time to set up the module XML file so you can replace the standard version with this one if you need to.

15.2.3 Setting up the property replacement

The `FlashMovie`'s module file contains only one entry: the directive to replace the standard implementation with the IE-specific one if the `user-agent` property is `ie6`. It's written as shown in listing 15.5.

Listing 15.5 Replacing `FlashMovieImpl` with `FlashMovieImplIE6` if the user agent is set to `ie6`

```
<module>
   <replace-with
            class="org.gwtbook.client.ui.impl.FlashMovieImplIE6">
      <when-type-is
            class="org.gwtbook.client.ui.impl.FlashMovieImpl"/>
      <when-property-is name="user.agent" value="ie6"/>
   </replace-with>
</module>
```

Now, when you create an instance of `FlashMovie`, you get an object that contains only the relevant tag necessary to display the movie in the browser you're using.

In addition to changing the application components based on the `user-agent` property, you can also manipulate the way GWT uses the standard internationalization functionality to change components based on the locale. But before you change components based on the locale, you need to fully understand how GWT manages internationalization.

15.3 Supporting internationalization in full

You've seen GWT's static i18n approach twice already in this book. In this section, we'll explain it in full, as well as looking at the dynamic approach you can use, which is particularly useful if you already have an existing i18n approach. In the static approach, you define a default properties file, say `Filename.properties`, and then define a number of locale-specific properties files whose filenames follow a predetermined structure, as shown in figure 15.3.

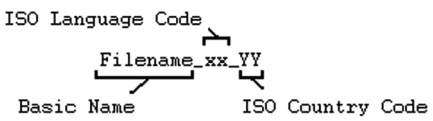

Figure 15.3 How locale-specific filenames are constructed

It isn't necessary to have an ISO country code, but if one is present, then the ISO language code must be, too. Locales also exist in a hierarchy, as shown in table 15.1.

Table 15.1 Permutation selected when using types that implement the `Localizable` interface

If the locale is set to	Then use this type
Unspecified value	The default type, `Type`
xx	The type called `Type_xx`, if it exists; or the default type `Type`, if `Type_xx` doesn't exist
xx_YY	The type called `Type_xx_YY`, if it exists; or the type called `Type_xx`, if that exists; or the default type `Type`, if neither `Type_xx_YY` nor `Type_xx` exists

In the Dashboard, you've already used the ISO language codes `en` for English and `sv` for Swedish; in the full Dashboard version, you'll also add a properties file for `en_US` (American English) where you provide the distinction in spelling of color from the `en` version: *colour*. This hierarchy has been chosen deliberately because GWT 1.4 introduces localization of times, dates, and currencies, and your choices of hierarchy could cause issues, as you'll see later.

First, we'll look at the static approaches you've already used in the Dashboard application.

15.3.1 Using static-string internationalization

The concept of string internationalization stems from the same principles used in standard code development. In normal code, it's common practice to avoid writing constants directly and instead to define the `String` literal as a constant in the code and then refer to that. You can also take it a step further by moving the constants out of the code and into a resource file. Table 15.2 shows each of these three ways, writing the constant directly, and defining and referring to the constant as a string literal, and defining the constant in a separate resource file.

Table 15.2 Different approaches to using strings in code

Approach	Code
Writing a constant directly in code	`Label newLabel = new Label("Some Label Text");`
Referring to a string literal in code	`final String labelText = "Some Label Text";` `Label newLabel = new Label(labelText);`

Referencing constants in a resource file	`MyConstants constants =`
	`  (MyConstants)GWT.create(MyConstants.class);`
	`Label newLabel = new Label(constants.labelText());`

The approach shown in the second row (defining a string literal constant first and then referring to it in the code) makes it easy to change the value of `labelText`, especially if it's used in numerous places in the code. You can take this a step further by moving the constants out of the code and into a resource file, referencing them from that file; this way, they're available across different classes. At this point, GWT static-string i18n can step in.

In the static approach, the compile uses generators that you saw in the last chapter to tie together an interface file and the variety of programmer provided properties files (one for each locale defined as in use). Because the GWT compiler creates permutations for each locale you identify by extending the `locale` property in the application's module XML file, it can employ the power of static analysis to include only those constants and messages that are used in the code. It also only includes code related to the specific locale for which the permutation is being created, further reducing the size of code per permutation, which translates to faster loading times for applications.

To implement static-string internationalization, you first provide a series of one or more properties files and a single Java interface that extends one of the following three GWT-provided Java interfaces:

- `Constants`—Simple user-interface constants. The compiler removes unused constants to improve efficiency.
- `ConstantsWithLookup`—Simple user-interface constants. The compiler keeps all provided key/value pairs. You can look up constants by name.
- `Message`—Simple user-interface messages that can take parameters, which are placed in predetermined locations of the message. The compiler removes unused constants to improve efficiency.

These three interfaces extend the `Localizable` interface, which is the link into the i18n generator, as we discussed in the last chapter. In this section, we'll look at these three interfaces and see how they're used. First is the `Constants` interface.

Defining static string internationalization constants

GWT i18n constants are simple strings that, as the name suggests, are constant—as opposed to messages that allow parameters to be displayed in the string. You need two types of files to implement i18n constants functionality in GWT, and we'll look at them in turn:

- A set of one or more properties files
- A Java interface that extends the `Constants` interface

The properties files are where the key-value pairs are defined. They're simple files that follow the naming convention shown for localized classes: There is a default file, say `myAppConstants.properties`, and you can define locale-specific files such as `myAppConstants_en.properties` and `myAppConstants_en_US.properties`. These files live in a hierarchy, and constants are searched for in that hierarchy in the same manner as indicated in table 15.1. For example, if the locale is set to `en_US` then a constant is searched for in the following order of properties files:

1. myAppConstants_en_US.properties
2. myAppConstants_en.properties
3. myAppConstants.properties

In the Dashboard application, you use this hierarchy (although the standard English text is kept in the default file) to cope with the different spelling of *colour/ color* between English and the version of English used in USA. Try it: When

the application loads, the Help menu contains an entry for the Colour Picker; if you change the locale to American, then that entry is spelled Color Picker.

This hierarchy also means you don't have to define all constants in all files. Look at the properties files shown in table 15.3. You can see the contents of a non-locale-specific properties file—`myAppConstants`—which provides no values for any constants apart from the `ok` key. There are also three location-specific properties files' contents—an English (`en`) locale `myAppConstants_en`, an American one, and a Swedish (`sv`) locale `myAppConstants_sv`. In these locale-specific files, language-specific values are defined for all keys except `ok`.

Table 15.3 The contents of three i18n properties files that exist in a hierarchy

myAppConstants	myAppConstants_en	myAppConstants_en_US	myAppConstants_sv
Colour: colour hello: hi: yes: no: ok:OK	hello: Hello there hi:Hi yes:Yes no:No	Colour:color	Colour: färg hello:Hejsan hi:Tjena yes:Ja no:Nej

You could now use the i18n tool provided in the GWT download to create the appropriate interface file (see chapter 3), or you could create it by hand. Either way, you would end up with an interface that looks as follows:

```
public interface MyAppConstants extends Constants{
   String hello();
   String hi();
   String yes();
   String no();
   String ok();
}
```

Note that this interface contains a method name for each of the constant keys in the properties file that you may wish to access—including the OK key. Regarding your part in coding the i18n setup, this is all you need to do; the GWT generator performs the task of creating necessary Java class files at runtime/compile time.

To use the Swedish, English and American locales in the application, you must extend the `locale` property using the module XML file by adding the appropriate `extend-property` tags:

```
<extend-property name="locale" values="en"/>
<extend-property name="locale" values="en_US"/>
<extend-property name="locale" values="sv"/>
```

You let the application know which locale to start with by setting a new `meta` tag in the head section with the name `gwt:property`; its content sets the locale of choice. Alternatively you can also change the locale by setting it as a parameter in the URL. For example, to use the Swedish version, the URL is

```
http://www.example.org/myapp.html?locale=sv
```

Look in the i18n module file (in the `com.google.gwt.i18n` package) to see how the system copes with changing the locale through URL parameters. In that file, a `property-provider` tag is defined to extract the URL and find the locale. This provides a good base code to use if you ever implement properties that are also changeable through URL parameters.

Finally, to use the different constants in the Java application code, you first need to obtain an independent version of the `MyAppConstants` class. Because this is an interface to your code, you use the `GWT.create()` method to indicate that you'll use the deferred binding approach to allow the compiler to decide the exact implementation at compile time. You do that by using the following code:

```
MyAppConstants constants =
     (MyAppConstants)GWT.create(MyAppConstants.class);
```

You use constants by calling the appropriate method defined in the interface:

```
Label newLabel = new Label(constants.hello());
```

The aim is to produce a menu that appears similar to figure 15.4.

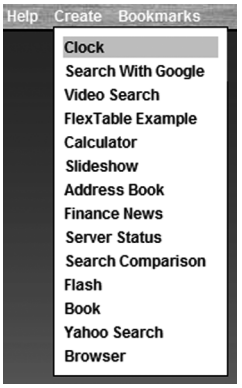

Figure 15.4 Menu system when the locale is set to the default

To create the text for the menu items in the default locale, you take the DashboardConstants.properties file previously created in chapter 3 and replace its contents with those shown in listing 15.6.

Listing 15.6 Defining the constants used in the default locale

```
HelpMenuName: Help
CreateMenuName: Create
AboutMenuItemName: About
LoginMenuItemName: Login
ConfirmDeleteMenuItemName: Confirm Delete
LocaleMenuItemName: Locale
CalculatorMenuItemName: Calculator
ClockMenuItemName: Clock
AddressBookMenuItemName: Address Book
```

Menu bar constants

Construct components

Menu item constants for Create menu

Menu item constants for Locale menu

```
SlideshowMenuItemName: Slideshow
GoogleSearchMenuItemName: Search With Google
GoogleVideoSearchMenuItemName: Video Search
FinanceNewsMenuItem: Finance News
ServerStatusMenuItem: Server Status
FlashMovieMenuItem: Flash
FlexTableMenuItem: FlexTable Example
SearchComparisonMenuItem: Search Comparison
BookMenuItem: Book
YahooSearchMenuItem: Yahoo Search
SendEmailMenuItem: Send Email
ColourPickerMenuItem: Colour Picker
NameChangeOK: Change
NameChangeCANCEL: Don't Change
EnglishLocale: English
SwedishLocale: Svenska
AmericanLocale: English (US)
CurrentLocale: en
```

Color constants

Text constants for EditableLabel buttons

See the following tip

TIP At runtime, it isn't directly possible to determine the locale currently used by the application (because the compilation process removes this information). However, if you really need to do this, you can write the locale string value as a constant in the properties files: for example, a `CurrentLocale` entry. Then, you can retrieve the value using the `constants.CurrentLocale()` method.

With the default locale established, you should add constants for the other locales you expect the Dashboard to manage.

You also need to replace the Swedish locale that you created in chapter 3 to match the entries in the default locale, but with Swedish text. (Don't forget that the encoding of the text file you're using should be set to UTF-8 if you're using a language's special characters.) Replace the `DashboardConstants_sv.properties` file's contents with those shown in listing 15.7, which produces a menu similar to that shown in figure 15.5.

Listing 15.7 Defining the constants used in the Swedish locale

```
HelpMenuName: Hjälp
CreateMenuName: Ny
AboutMenuItemName: Om Dashboard
ConfirmDeleteMenuItemName: Godkännar Delete
LocaleMenuItemName: Välj Språk
CalculatorMenuItemName: Kalkylator
ClockMenuItemName: Klocka
AddressBookMenuItemName: Adressbok
```

Menu bar constants

Menu item constants for Create menu

Menu item constants for Locale menu

```
SlideshowMenuItemName: Galleri
GoogleSearchMenuItemName: Sök med Google
GoogleVideoSearchMenuItemName: Sök Video
FinanceNewsMenuItem: Finans Nyheter
SearchComparisonMenuItem: Search Comparison
BookMenuItem: Boken
YahooSearchMenuItem: Sök med Yahoo
SendEmailMenuItem: Skicka en Email
ColourPickerMenuItem: Välj Färg
NameChangeOK: Byta
NameChangeCANCEL: Ej Byta
CurrentLocale: sv
```

Color constants

Text constants for EditableLabel buttons

We made a subtle "error" in the properties file for the Swedish locale in order to discuss a specific attribute of GWT i18n. If you compare the default and Swedish locales, you'll see that the Swedish properties file misses a number of constants. When GWT comes across this situation, it simply goes to the next level in the hierarchy to find the constant value (in this case the default locale—which explains the English words visible in figure 15.5.)

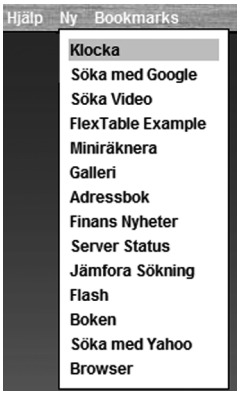

Figure 15.5 Menu system when the locale is set to Swedish by changing the `DashboardConstants_sv.properties` file's contents

Finally, you create the American English properties file (`DashboardConstant_en_US.properties`) which makes heavy use of the hierarchy to only define the following two lines:

```
ColourPickerMenuItem: Color Picker
CurrentLocale: en_US
```

If you want to add additional locales, now is a good time to do so. Remember that the filename must follow the format specified in chapter 15 and that you also need to add an `<extend-property>` tag to the Dashboard.gwt.xml file defining each locale you're adding (if you've not already done so, then you need to add the `en_US` locale to the Dashboard.gwt.xml file).

Once you're happy that your default properties file contains all the constants you need in the application, then you need to execute the `DashboardConstants-i18n` tool again (remember that this tool was created in chapter 2). It regenerates the necessary `DashboardConstants` interface file, taking into account any changes you've made since the last time it was executed. The result of executing the tool for the default file is shown in listing 15.8.

Listing 15.8 Result of the `DashboardConstants-i18n` tool

```
public interface DashboardConstants extends
                 com.google.gwt.i18n.client.Constants {

  String AboutMenuItemName();
  String CreateMenuName();
  String HelpMenuName();
  String CalculatorMenuItemName();
  String SlideshowMenuItemName();
  String AddressBookMenuItemName();
  // other signatures
  String NameChangeOK();
  String NameChangeCancel();
  String SwedishLocale();
  String EnglishLocale();
  String CurrentLocale();
}
```

In addition to a `Constants` interface, GWT provides a `ConstantsWithLookup` interface, which we'll look at next. The main difference is that `ConstantsWithLookup` doesn't perform any static code reduction.

Defining static string internationlization constants with lookup

The `ConstantsWithLookup` interface works exactly the same way as the `Constants` interface, and values in the properties file can be accessed in exactly the same way. What sets these two apart is that `ConstantsWithLookup` sends all constants into the JavaScript file (so there's no reduction in file size), and it additionally provides a number of specific retrieval methods that return the constant as a variety of Java objects. These retrieval methods are as follows:

- `getBoolean(String)`—Looks up a particular method name, and returns a `boolean` value
- `getDouble(String)`—Looks up a particular method name, and returns a `double` value
- `getFloat(String)`—Looks up a particular method name, and returns a `float` value
- `getInt(String)`—Looks up a particular method name, and returns a `int` value
- `getMap(String)`—Looks up a particular method name, and returns a `map` value

- `getString(String)`—Looks up a particular method name, and returns a `String` value
- `getStringArray(String)`—Looks up a particular method name, and returns a `String[]` value

An exception is raised if the method name doesn't exist in your interface or if it isn't possible to cast the constant type to the object expected (for example, a `String` to a `boolean`). If you define the interface as follows

```
public interface MyAppConstants extends ConstantsWithLookup{
   String hello();
   String hi();
   String yes();
   String no();
   String ok();
}
```

then you can look up values from your properties files using one of the following two approaches:

```
constants.hello();
```

or

```
constants.getString("hello");
```

If you tried to look up `constants.getString("farewell")`, it would throw a runtime exception because the method name `farewell()` doesn't exist (if you had used the static approach, this error would have been picked up at compile time). Similarly, looking up `constants.getInt("ok")` would throw a runtime exception because the `ok()` method returns a `String` and not an `int`.

Let's get even racier with our examples and start defining i18n messages—something you didn't do earlier—that you'll use in the Dashboard application. In addition to constants, you can use messages in the same style by extending the `Messages` interface. Messages allow parameters to be passed in the interface; the values of those parameters appear in predetermined sections of the messages. If you're using the GWT creation tools to create the files, don't forget to include the `-createMessages` flag when you're running the `i18nCreator` tool. In the next section, we'll look at how these messages are created.

Defining static-string internationalization messages

Messages differ from constants in that they're parameterized. As with constants, you need two types of files:

- A set of one or more properties files
- A Java interface that extends the `Messages` interface

The properties files are again where the key-value pairs are defined, and they follow the same naming convention and hierarchical properties as those discussed for constants. As an example, you can define `myAppMessages.properties` file as the following:

```
hello:Hello {0},
```

Defining a message is like creating a template where values will be slotted in. The example Hello message has one slot, denoted by the text `{0}`. Other messages may contain more slots: for example, `goodbye:Goodbye {0} at {1}`. This message has two slots; in the interface, you define what goes into them. You can, of course, define messages with zero slots, in which case they act like constants in the way we described earlier.

The interface you need to define for messages is almost the same as for constants, except that you should extend the `Messages` interface, and you define the parameters the methods should take—one parameter for each slot in the message. For the properties file example, you define the interface method as follows:

```
public interface MyAppMessages extends Messages{
   String hello(String name);
}
```

You're saying that the `hello()` method takes one parameter, which is a `String`.

Using the code in the application and setting the locale is performed the same way as for the other two techniques we've just discussed, except that you need to pass the appropriate number of parameters to the chosen method. For example, to display the message "Hello Tiina Harkonen", you write the following code:

```
MyAppMessage messages = (MyAppMessage)GWT.create(MyAppMessage.class);
Label newLabel = new Label(messages.hello("Tiina Harkonen"));
```

Other specific locale properties files are created the same way as for the other interfaces we've looked at. For example, a properties file for the Finnish locale, `fi`, is called `myAppMessages_fi.properties` and may have the contents

```
hello: Tervetuloa {0},
```

If the application was set to the Finnish locale, then the hello message would become "Tervetuloa Tiina Harkonen".

In chapter 2, we discussed that the `i18nCreator` tool can be used to create messages as well as constants (messages allow you to add a parameterized value

into the text at runtime, whereas constants don't). You'll now bring in a message that you'll use to set the default name on the Dashboard (which is stored in the `EditableLabel`). The first thing to do is to create the necessary structure and files using the `i18nCreator` tool.

Establishing the messages file and directory structure

To create the messages structure, you should execute the appropriate command line shown in table 15.4, depending on whether you're using the Eclipse IDE (this will work, assuming you've followed the same structure you created in chapter 2).

Table 15.4 Different versions of the `i18nCreator` tool used to create the message framework for the Dashboard internationalization

Version	Command Line
Non Eclipse	`i18nCreator -createMessages -out DashboardDir org.gwtbook.client.DashboardMessages`
Eclipse	`i18nCreator -eclipse DashboardPrj -createMessages -out DashboardDir org.gwtbook.client.DashboardMessages`

Successful execution of the command results in a new DashboardMessages.properties file in the code directory and a new application called `DashboardMessages-i18n`. (If you use the Eclipse version, you also get a new Eclipse launch configuration. In Eclipse, you should now refresh your project by right-clicking it and selecting the Refresh option; doing so shows these new files in your Package Explorer view.)

Now that you've created the message structure, let's create the default locale implementation.

Creating messages for the default locale

Just like creating constants for the default locale, creating messages for the default locale means you must update the `DashboardMessages.properties` file created by the `i18nCreator` tool. Find it now, and replace it with the contents shown in listing 15.9.

Listing 15.9 Part of the default DashboardMessages.properties file

```
DashboardDefaultNameMessage = New Dashboard Created at {0}      (1)
ConfirmDeleteMessage = Are you sure you want to delete the      (2)
                       {0} application?
BookmarkOnErrorMessage = Some problem has occurred, we are unable to
                         load your bookmarks.
```

```
BookmarkOnResponseErrorMessage = Some problem has occurred, the
                                 server returned a status code {0}
                                 (perhaps you are running this with
                                 no server?)
WindowResizedMessage = New window size: ({0},{1}) \n Turn Off Resize
                       Notifications?                                  ❸
SaveDashboardNameMessage = We would normally save the \n Dashboard
                           Name now it is changed.
WindowClosingText: You are attempting to close the Window - are you
                   sure?
WindowClosedText: Window is now closing, typically at this point
                  we would now save the state of the application.      ❹
```

Notice that in the messages, you use the equals sign (=) as well as the colon (:) to separate the keys from the message; this is just to show the flexibility we mentioned earlier. At ❶ is a message that takes two parameters: in this case, the x and y dimensions of the resized window.

This code creates a set of GWT i18n messages, each with a differing style. The first line ❶ places the variable text at the end of the message (you'll use this in the Dashboard as the initial text for the `EditableLabel` that represents the Dashboard's name). It's simple to place the variable text in the middle of the message, as shown on the second line ❷, or it can go at the start. Placing two or more variables in the message? Not a problem; you do that at ❸. Finally, you hit a slightly philosophical point where there are no variables to add to a message—should it be a message or a constant? Technically, it should be a constant, but in the last line, ❹, it's a message. Where should it be placed? In professional development, you can always refer back to your coding standards for guidance (or, if it isn't there, create it). For the Dashboard, we decided that where the text is presented to the user as a message, it should sit in the messages area regardless of whether it has variables.

If you execute the new `DashboardMessages-i18n` command (created by the `i18nCreator` tool), it creates a new interface file called `DashboardMessages.java`. The interface's contents resemble listing 15.10 (you will explicitly set the parameter type for the `WindowResizedMessage` to `int`).

Listing 15.10 Results of the `DashboardMessages-i18n` tool

```
public interface DashboardMessages extends
   com.google.gwt.i18n.client.Messages {
    public String DashboardDefaultNameMessage(String time);
    public String ConfirmDeleteMessage(String appName);
```

Message with one String variable

```
    public String WindowResizedMessage(int x, int y);
    public String WindowClosingText();
    // Other signatures
}
```

Message with two int variables

Message with no variables

The resulting Java interface is similar to the constants interface, except the method takes a parameter, the value of which is inserted in the place-marker `{0}` defined in the message.

You must add similar messages for the different locales the application will manage.

Adding messages for other locales

GWT's i18n approach is standard. To create different locale messages, you create different properties files named according to the same naming convention used for constants. In this case, you create a DashboardMessages_sv.properties file for the Swedish locale and place the following in it:

```
DashboardDefaultNameMessage = Ny Dashboard, frambringade @ {0}
```

But you needn't stop with localizing constants and messages; you can do the same thing with components in the application. For example, the trash icon changes based on the locale. We'll explain how you can alter components based on the locale in more detail in section 15.4.

One of the restrictions of using a static approach is that changing locale requires you to reload your GWT application. For small applications, this may not be a major problem; but if your application becomes substantial in size, then a simple locale change may begin frustrating your user. However, the static nature which finds errors at compile time and reduces code size should usually outweigh the reloading problem.

GWT (version 1.4) also provides a number of classes that deal with localization of dates, times, and currencies.

Localization of dates, times, and currencies

When you create the message that is the default value for the `EditableLabel` (in the Dashboard's `onModuleLoad()` method), you use the `DashboardDefaultNameMessage` from your messages files to display the date and time the Dashboard was created.

To do so, you use the `DateTimeFormat` i18n classes provided from GWT 1.4. You create two new objects, one that creates a full date format

```
String fullDateFormat =
       DateTimeFormat.getFullDateFormat().format(new Date());
```

and one that creates a short time format:

```
String shortTimeFormat =
       DateTimeFormat.getShortTimeFormat().format(new Date());
```

In the `com.google.gwt.i18n.client.constants` package, you can find hundreds of classes covering dates and number formats for all the potential language and country codes. The date format object, for example, provides details about the `short` and `full` date/time formats. Listing 15.11 gives the details of the default English locale properties file for the `DateTimeFormat`.

Listing 15.11 DateTimeConstants_en properties file

```
eras = BC, AD
eraNames = Before Christ, Anno Domini
narrowMonths = J, F, M, A, M, J, J, A, S, O, N, D
months = January, February, March, April, May, June, July, August,
         September, October, November, December
shortMonths = Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,
              Nov, Dec
// More month data
weekdays = Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
           Saturday
shortWeekdays = Sun, Mon, Tue, Wed, Thu, Fri, Sat
narrowWeekdays = S, M, T, W, T, F, S
// More weekday data
ampms = AM, PM
dateFormats = EEEE\\, MMMM d\\, yyyy, MMMM d\\, yyyy, MMM d\\,
              yyyy, M/d/yy
timeFormats = h:mm:ss a v, h:mm:ss a z, h:mm:ss a, h:mm a
```

These definitions in the locale-specific files are used by the `DateTimeFormat` class the same way you've been using i18n properties files to ensure the appropriate set of definitions is used for your locale. You use the `DateTimeFormat` class to get text representing dates and times in the format you desire. In the Dashboard, you use the following code to create the `EditableLabel`:

```
dashboardName = new EditableLabel(
                    messages.DashboardDefaultNameMessage(
                          medTimeFormat +" "+ fullDateFormat),
                    constants.NameChangeOK(),
                    constants.NameChangeCANCEL());
```

You tie together the GWT `DateTimeFormat` classes to get the presentation of dates and times for your locale (for example, the U.S. uses a 12-hour clock as opposed

to the 24-hour clock used in Europe). In the U.S. locale, the variable `medTimeFormat` might hold

```
03:02 PM
```

whereas in the English and Swedish locales, it would hold

```
15:02
```

However, here is the subtle problem in the i18n approach applied so far. The default English locale in the Dashboard is English, and we specifically included an American English locale (`en_US`) to resolve the spelling of *colour/color*. When you include GWT's date and time formats, then they define the default English locale as American English—and you have a mismatch. In the Dashboard's default English locale, the spelling of *color* is English (*colour*), but the time format uses the 12-hour clock rather than the expected 24-hour clock.

Correct or incorrect is a debate that could go on for a long time, but you should be wary of this in your applications, especially if you're developing from a non-American viewpoint (it isn't just a debate over English primacy—the default Spanish locale, as another example, is Argentinian). Be particularly careful if you're using currencies, because, for example, the default English currency is U.S. dollars, and the default Spanish currency is Argentinian Pesos (unless GWT changes this in the future). To be 100 percent safe, if you're using currency/date/time formats, you may want to avoid default locales and specifically use both language and country codes.

We've exhausted the static approaches; they're the ones you should use unless you have an existing i18n approach. Whether you use constants, constants with lookup, or messages will depend on your application.

Dynamic-string i18n is the other approach that GWT provides; it's a more flexible approach to i18n with regard to changing locales, although this flexibility comes at the expense of static analysis. The dynamic approach also lets you use existing approaches to i18n that you may already have in the rest of your site/organization. We'll now look at this dynamic approach.

15.3.2 Using dynamic string internationalization

The dynamic approach was originally designed to allow existing i18n approaches to be quickly incorporated into GWT applications. If your existing approach used JavaScript associative array objects containing sets of key-value pairs, like those shown in table 15.5, then dynamic-string i18n would potentially work for you.

Table 15.5 Two JavaScript associative array objects containing English and Swedish user interface constants

`var userInterfaceEN = {`
`  hello:Hello,`
`  hi:Hello,`
`  yes:Yes,`
`  no:No,`
`  ok:OK`
`};`
	`var userInterfaceSV = {`
	`  hello:Hejsan,`
	`  hi:Tjena,`
	`  yes:Ja,`
	`  no:Nej,`
	`  ok:OK`
	`};`

In this approach, the two JavaScript objects list key-value pairs for a simple user interface, the first in English and the second in Swedish. In JavaScript, it's easy to select an element in an associative array by referencing it via the key. To select the correct text to display for the index `hello` in a Swedish locale, you write

```
userInterfaceSV[hello];
```

which results in the text "Hejsan" being selected.

If this is the approach you currently use for i18n, then implementing it in GWT is swift and easy. You need to insert the JavaScript objects into the HTML file as you normally would; they're accessed using the GWT `Dictionary` class.

Assuming the HTML file into which the GWT application is to be loaded has the `userInterfaceSV` associative array included, then you create an object that accesses it using the `getDictionary()` method, as follows:

```
Dictionary uiText = Dictionary.getDictionary("userInterfaceSV");
```

The `Dictionary.get(key)` method retrieves values based on the supplied key. In this case, to retrieve the value to display for the index `hello` in a Swedish locale, you can write

```
uiText.get("hello")
```

Dynamic-string i18n offers a few benefits:

- You can quickly use existing i18n approaches you currently have.
- No code recompilation is required to make any changes or additions to the constants.
- Changing locale doesn't necessarily require a complete reload of the application.

However, the disadvantage of the dynamic approach is that GWT provides no help to determine whether constants you're referring to exist. With the dynamic

approach, it's possible to refer to keys that don't exist, creating unexpected results in your UI. In addition, the compiler can't perform any optimization by removing unused keys. This means all key-value pairs, used or not, are sent by the web server, increasing the application's size and, therefore, slowing down the delivery of applications.

GWT's i18n approach is great for dealing with the display of different messages and constants based on locales, but you can go one better and start changing complete components of the application based on the locale.

15.4 Altering the application for the locale

Let's imagine that one of the Dashboard applications will access restricted information. That information could be anything, but for the sake of the example, we'll say that it's financial information for a company that is restricted by financial market regulations. Although GWT's i18n approach is set up for dealing with messages and constants, you can subvert it to change application components based on locale by borrowing the code structure.

You need to implement two basic types to get the differing functionality by locale (this pattern should be becoming familiar to you now!):

- A default class
- A set of zero or more localized classes

Let's look at both cases in more detail through examining the Dashboard Finance News component application.

15.4.1 Implementing the default component

When companies make certain announcements, it's common for this information to be restricted to certain financial markets due to regulations. For example, the details of one company intending to purchase a controlling stake in another company listed on the UK market isn't usually for release in the USA or Australia. You can use localization to restrict this access.

For the financial system application's functionality, you want to display a button allowing access to the announcement if a user is in a locale that can see the announcement and a label expressing regret if not. As a starting point, you'll define the default type, in this case a class, and provide a single method that returns a GWT widget. Because you wish to be safe and not attract the wrath of regulatory bodies, you want users coming in from any locale to be treated to a display of the default screen shown in figure 15.6.

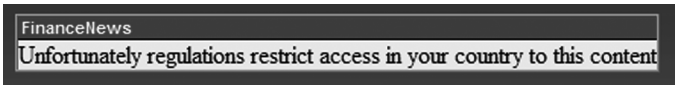

Figure 15.6 Finance application running in a locale that isn't allowed to view financial information

To do this, you set the default class to return a `Label` containing denial message. Call the class `Proceed`, and write the following:

```
public class Proceed implements Localizable{
   public Widget getProceed(){
      return new Label("Unfortunately regulations restrict
                        access in your country to this content");
   }
}
```

The default class is simple and has the basic rule that it must implement the `Localizable` interface. Let's move on to the locale-specific classes.

15.4.2 Locale-specific classes

Let's say that a financial announcement is made on the Swedish stock market. Users in the Swedish locale can read the text, and instead of the previous message, you want to present a button allowing the user to click through to the announcement.

In theory, a user in the Swedish locale will see the full release after they click on the proceed button, but for this example, you will just show them an alert box as shown in figure 15.7.

You distinguish users from the Swedish locale by defining that the locale they're in has the ISO language code `sv`. You want to write a class that is used for this locale instead of the default class, and you want this new class to return a button rather than a label. To do so, you write a class that extends the original `Proceed` class and is named following a simple naming convention. Here it is:

Figure 15.7 Finance application in `sv` locale (even though the text is written in English so the majority of users can see the functionality)

```
public class Proceed_sv extends Proceed{
   public Widget getProceed(){
      Button theButton = new Button
                      ("Please press to Proceed to the News Item");
      theButton.addClickListener(new ClickListener(){
         public void onClick(Widget sender) {
            Window.alert("Going to News Article");
         }
      });
      return theButton;
   }
}
```

That is all you need to do from a code perspective to implement localization classes in a GWT application. Unfortunately, it doesn't mean your localized code is available to your application—that takes a couple more steps. The first step is to tell the application which locales it needs to be concerned about, and the second is to tell it which locale to use (you already did this for the Swedish locale in the Dashboard application some chapters back).

Unlike with browser-specific code, when you replace components due to locale, there is no need to enter anything beyond the locales in the module XML file. The existing setup provided by GWT in i18n's module XML file is sufficient.

Now that you've seen how to use the existing approaches for properties with your code, it's time to take the next step and begin defining and managing new properties.

15.5 Implementing user-defined properties

You've seen that GWT provides a nice way to manage different browsers through properties; it also gives you a way to manage different locales for messages and constants, which you can subvert to manage the changing of application components based on locales. It's possible to take this property-based approach further and define your own properties and handling code.

In this section, you'll do that for the Dashboard application to present an intranet view and a more restricted Internet view. In your applications, you may need to do something similar so that external users get a restricted set of functionality compared to users on your intranet. There are many ways to provide this type of division outside of GWT; this is just an approach that you may want to investigate. (You set the property in the HTML file, which isn't that secure because the user can override it; but you can expand the property-provider approach and generate JavaScript that selects which version the user sees based on IP addresses or something similar.)

15.5.1 Defining user-specified properties

The property definition is simple. You'll define a user-defined property that has two values—`intranet` and `internet`—and you'll call it `externalvisibility`. You set this up using the standard approach that we described a few sections back, by making the following entry in the Dashboard.gwt.xml file:

```
<define-property name="externalvisibility" values="intranet,internet"/>
```

Before it can be used in any meaningful way, you need to be able to get the value of this property.

15.5.2 Defining a user-specified property provider

In the Dashboard example, you look at the `meta` tag defined in the HTML file to determine the start value for the external visibility flag. More realistically, you might implement IP-checking code to set the value, but we'll stick with getting it from the HTML file. You do that in the property provider shown in listing 15.12, which is placed in the Dashboard's module XML file.

Listing 15.12 Defining the property-provider that handles your `externalvisibility` property

```
<property-provider name="externalvisibility">
<![CDATA[
   try{
      var externalvisibility =
             __gwt_getMetaProperty("externalvisibility");
      if (externalvisibility==null){
         externalvisibility = "internet";
      }
      return externalvisibility;
   } catch (e) {
      return "internet";
   }
]]>
</property-provider>
```

Access HTML meta definition ❶

Set default property value ❷

Return value ❸

Return default value in case of error ❹

In the property provider, you first try to get the `externalvisibilty` value from the `meta` tag in the Dashboard.html file using the `getMetaProperty()` method ❶. If that isn't available, then an exception is raised, which you catch at ❹; you then set the value to the most restrictive value, `internet`. If you successfully get a value at ❶, then you check to make sure it isn't null ❷; if it is, you set it to be the value

internet; otherwise, you return the value that you found ❸. How do you know the value is OK? We'll look at that next.

15.5.3 Checking the provided property value

Just retrieving a value from the metatag doesn't necessarily mean it's a valid value that you can work with. Fortunately, you can harness GWT again to ensure the value is in the list of defined properties—or, if it isn't, to handle that error.

In the Dashboard.html file, you define a `gwt:onPropertyErrorFn` `meta` tag as follows:

```
<meta name='gwt:onPropertyErrorFn' content='handleWrongVisibility'>     ❶
   <script>
      function handleWrongVisibility(propName,
                                     allowedValues,     ❷
                                     badValue){
         if (propName == "externalvisibility"){     ❸
            window.alert("You are looking at the application from
                          an unknown area\nCheck the
                          externalvisibility property.");
            window.location.href = ("http://www.google.se");     ❹
         }
      }
   </script>
```

Here you define the `meta` tag ❶ followed by some JavaScript that is executed when GWT determines that the property value given to the application is invalid. You indicate in the `meta` tag that the function `handleWrongVisibility` should be called if there is a property error. At ❷, you begin to provide the definition of the `handleWrongVisibility` method. At ❸, you check whether the error is with the `externalvisibility` property, and if so, display an error and navigate the user to the Google home page ❹. You can extend this approach to all the properties you have, if you wish.

With the basics in place, you can now build the code.

15.5.4 Building the code

Finally, you can build the complete final version of the Dashboard, which comes in two flavors: one for the Internet and one for the intranet. The Internet version provides access a small number of component applications; the intranet version provides those same component applications plus many more.

You implement this by following the same pattern you have for all the property-based functions. First, you build the `Internet` class as a default class for the property, which you'll call `Dashboard`. In this `Dashboard` class, you create

the menus containing the limited number of menu items corresponding to component applications. Next, you build a `Dashboard_intranet` class that extends that `Dashboard` class and overrides the methods in `Dashboard` that are responsible for building the menus (and therefore giving access to the additional component applications).

To complete the functionality, add a `replace-with` tag in the Dashboard's module XML file:

```
<replace-with class="org.gwtbook.client.Dashboard_intranet">
   <when-type-is class="org.gwtbook.client.Dashboard"/>
   <when-property-is name="externalvisibility" value="intranet"/>
</replace-with>
```

This entry says that the `Dashboard` class must be replaced by the `Dashboard_intranet` class if the `externalvisibility` property is set to `intranet`.

15.6 Summary

This concludes our walk through using properties to change the application—a powerful tool, particularly if you need to alter aspects of your application to suit differing locales. Just remember that the general pattern is to create a default class and then the variations, all which extend the default class. Then, you can use `replace-with` tags in the module XML file to replace the default file when properties match values; or, if you're using the i18n approach, the default class must implement `Localizable`, and all class names should follow the i18n naming structure.

In the next part of this book, we'll look at the final practical aspects of GWT, including testing and deploying applications.