Sample Chapter

Robert T. Cooper
Charlie E. Collins

GWT IN PRACTICE

- 39 TECHNIQUES
- GET IT DONE
- GET SAVVY

MANNING

GWT in Practice

by Robert Cooper and Charles Collins

Sample Chapter 6

6 Integrating Legacy and Third-Party Ajax Libraries

This chapter covers

- Working with the JavaScript Native Interface
- Creating JSNI Wrappers
- Dealing with JavaScript eventing models
- Working with the JSIO project

Before software can be reusable it first has to be usable.

—Ralph Johnson

Although we believe GWT represents a superior environment for developing Ajax applications, sometimes the functionality you want to reuse is in existing JavaScript libraries. These might be your own legacy code or third-party libraries. Additionally, there may be times when you want to directly tweak aspects of your GWT application with JavaScript. To perform either of these tasks, you need to use JSNI.

We introduced JSNI in chapter 1 and used it in several other examples in this book, so you should already have a handle on the basics. Though JSNI is very powerful, it can also be confusing and problematic. When working with JSNI, you have to keep in mind that you're on your own, and your own feet are in range of your

shotgun. Methods that you implement as native JavaScript, or that you wrap and inherit, are typically less portable across different browsers, are more prone to memory leaks, and are less performant than JavaScript optimized and emitted by the GWT compiler. For this reason, though it's possible to use JSNI for more than integration, we recommend that you stick to integrating existing well-tested JavaScript code when using JSNI. That will be our focus here.

In this chapter, we'll take a closer look at the specifics of JSNI usage and at some common gotchas you may encounter. Knowing a few key things can lessen the confusion and reduce the possibility for errors. Then we'll look at a couple of larger examples where we take two popular JavaScript libraries, Script.aculo.us and Moo.fx, and integrate them with GWT by creating a wrapper module for each.

As we present these examples, you'll see common tasks and patterns for working with JavaScript from GWT, including packaging JavaScript files, dealing with typing compatibility, and handling events. We'll begin with a recap of JSNI basics and then move into more involved examples.

6.1 A closer look at JSNI

JSNI is required any time you want to call into native JavaScript from Java, or vice-versa, so you need to understand it when working with any existing JavaScript or when tweaking display parameters beyond the exposure of the GWT browser abstraction. The GWT documentation notes that JSNI can be thought of as the "web equivalent of inline assembly code." This goes back to the browser-as-virtual-machine idea that GWT embraces.

Before we dive into some involved GWT "assembly" code, we'll revisit the JSNI basics, cover some common pitfalls, and get our working environment for this chapter's examples in order.

6.1.1 JSNI basics revisited

There are two sides to the JSNI coin: Java and JavaScript.

You can write native JavaScript methods directly in Java by using the `native` keyword and the special `/*- -*/` comment syntax. This syntax gives you the opportunity to write JavaScript code that will be run in the browser, as is. Here's an example:

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

You can also access Java fields and methods from within JavaScript by using another special syntax that indicates classes with the `@` symbol and method or field names with `::` (two colons). Examples of each of these follow (where `instance-expr` is optional and represents either an instance if present or a static reference if absent):

```
[instance-expr.]@class-name::field-name

[instance-expr.]@class-name::method-name(param-signature)(arguments)
```

Along with having a unique syntax, JSNI also has a specific type mapping. Like JNI does with platform-specific code, GWT uses a set of character tokens to indicate the type information for the method you're calling. Understanding this is important, because type conversion will determine what parameters are being passed and *which* method will be called in the case of overloaded methods. Table 6.1 lists the tokens for each of the standard Java types and how they're mapped using JSNI tokens.

For example, if you had a Java method with a signature like the following,

```
public String figureSomething(String[] string, int startPoint, boolean
  flag);
```

you would invoke it from JSNI with a statement such as this:

```
x.@com.my.ClassName:: figureSomething([Ljava/lang/String;IZ)(array, 2,
  false);
```

Notice that there is no separation between the tokens in the JSNI arguments portion.

Another important basic component in JSNI code is the `JavaScriptObject` class. We used this magical class in our examples in chapter 5 as the return type of our native methods. That's the basic job of `JavaScriptObject`. It provides an opaque representation of a JavaScript result to Java.

While the JSNI rules seem simple, troubleshooting can sometimes be problematic. When incorporating legacy JavaScript, it's critical that you understand the potential pitfalls of using the JSNI libraries.

Table 6.1 Mappings of Java types to JSNI type signature tokens

Java type	JSNI signature token
`boolean`	`Z`
`byte`	`B`
`char`	`C`
`short`	`S`
`int`	`I`
`long`	`J`
`float`	`F`
`double`	`D`
Any Java class	`L(fully-qualified class path);` example: `Ljava/lang/String;`
Arrays	`[` followed by the JSNI token for the contained type

6.1.2 Potential JSNI pitfalls

The dual nature of JSNI makes things a bit more complicated than traditional coding. You have to be aware of not only the logic and usage of one language, but also of the syntax, interaction, type conversions, and boundaries with another. Several potentially problematic issues are often encountered.

It's important to remember that arrays of primitives or `JavaScriptObject`s passed into a native method will be completely opaque and can only be passed back via JSNI or returned. Additionally, a JavaScript array of string values can't be passed as `Ljava/lang/String;`. You'll see an example of how to get around this in section 6.3.4.

As you saw in chapter 5, `JavaScriptObject` modifies the native object as it's passed into Java code. `JavaScriptObject` is instrumented with functions like `equals()` and `hashCode()` to make it easier to work with in Java. If you're using a security-restricted object, like a DOM element representing a plugin or ActiveX control, this will cause security exceptions. Also, `JavaScriptObject`s denoting the same native reference will not evaluate to `true` with the `==` operator. You must use the `equals()` method.

Another important thing to keep in mind when you're working with JSNI code is the behavior of the `JavaScriptObject`. You should never return the JavaScript special value `undefined` to your Java code, as it can lead to all kinds of unexpected results. If you do, your compiled Java code will continue to pass this around until there is a call to a utility method, or something that evaluates one of the GWT-added values, and strange `HostedModeException`s will start to be thrown. If you do this, you can get lost for hours trying to figure out where things went wrong.

While including JavaScript in your Java code is very easy with JSNI, sometimes you just want to include a JavaScript file. For instance, if you have a third-party JavaScript library, or if you're using your own legacy JavaScript, you need to make those files available to your JSNI code inside your GWT application. The easiest way to do this is to use a `<script>` tag in your module's gwt.xml file. While you can link scripts directly from host pages, doing so can be problematic—modules cannot be easily distributed and reused if they need an external JavaScript file.

As we discussed in chapter 2, the `<script>` tag in your module definition takes care of this by allowing a module definition to include a JavaScript file in the public package of the module, and it tells the gwt.js file to load that script before the rest of the module starts executing. Since the public directory will be included in any modules that inherit from the original, the loading of the script is automatic. Listing 6.1 shows a sample.

Listing 6.1 A simple use of the `<script>` tag

```
<module>
    <script src="myscript.js"><![CDATA[
        if( $wnd.MyClass ) {
            return true;
        } else {
```

```
            return false;
        }
    ]]></script>
</module>
```

The `src` attribute is almost self-explanatory. This is the location of the JavaScript file relative to the public package of the module. Inside the `<script>` tag is a block of code that can be executed, returning `true` or `false`, to determine whether the script has been successfully loaded. This is JavaScript written in JSNI notation, and it is usually contained in a `CDATA` section because, well, because entity-escaped code is ugly. You'll also notice the use of the `$wnd` global variable. We'll get to that in a bit.

Why do we need this block of code? Although including a script in the HTML harness page is easy, the script has to work within the confines of the single-threaded JavaScript interpreter for GWT to load it. Even a script that's being included dynamically has to be executed on the same thread as the GWT application.

Let's say you wanted to write a method to load a script file. You might write something like this:

```
private void addScriptTag(String source) {
    Element script = DOM.createElement("script");
    DOM.setAttribute(script, "src", source);
    DOM.appendChild(RootPanel.getBodyElement(), script);
}
```

Using this technique, you would find out very quickly that your application behaves erratically. Why? Because that script will not execute at the moment `DOM.appendChild()` is called. It will queue up and wait for your GWT application to go into event-wait before execution, like Glick's metaphorical bees waiting to visit the queen. Therefore, any method you called in the same execution run as `addScriptTag()` would never see the script. You could, of course, create an event callback to let you know when the script is loaded—and if you wanted to use a JSON service dynamically within JSNI from a non-same-origin server, this might be a good idea. For most uses, though, this is a lot of code to write when what you really want is for the script to be available when your GWT module begins executing.

Going back to listing 6.1, the `$wnd` global variable is one of two special tokens GWT supports inside JSNI code, the other being `$doc`. The entirety of your monolithic GWT application runs in an isolated context, so the usual JavaScript global variables, `window` and `document`, will not give you access to the HTML host page's window and document values. To access them, you need to use `$wnd` and `$doc` directly. Scripts that are loaded with a `<script>` tag are in the host page's execution context. This means that if you want to call methods or use objects defined in your external scripts, you need to prefix these calls with `$wnd` to make sure that GWT calls the top level and not the GWT scope.

Now that we have covered the specifics of talking to JavaScript and including JavaScript in your GWT application and we've addressed some common issues, we'll look at some examples where we wrap third-party JavaScript for use in GWT. First,

though, we need to set up IntelliJ IDEA, our spotlight tool in this chapter, for use with GWT.

6.1.3 Configuring IntelliJ IDEA

It's not coincidental that this chapter on integrating JavaScript using JSNI is the one in which we're introducing IntelliJ IDEA as the spotlight tool. Of all the IDEs, IDEA's core GWT support is exceptionally good. While it's not a free option like Eclipse or NetBeans, it's a fantastic development environment. For GWT applications, it includes a lot of shortcuts for creating modules, enhanced highlighting, and more.

To get started with IDEA, you must first configure the GWT home location, as you do for the NetBeans module. This is done by selecting the Settings option (the wrench icon on the toolbar) and selecting the GWT icon, as shown in figure 6.1.

Clicking this icon brings up a dialog box prompting you for the location of the GWT home directory. At we write this, IDEA doesn't know about GWT for the Macintosh, but that will likely change soon. You can sidestep the error message on the Mac by simply copying and renaming the gwt-dev-mac.jar file in the home folder to gwt-dev-linux.jar.

What makes IDEA's support special? Well, two things stand out as great features. First, it restricts your project's classpath scanning to modules declared as `<inherits>`

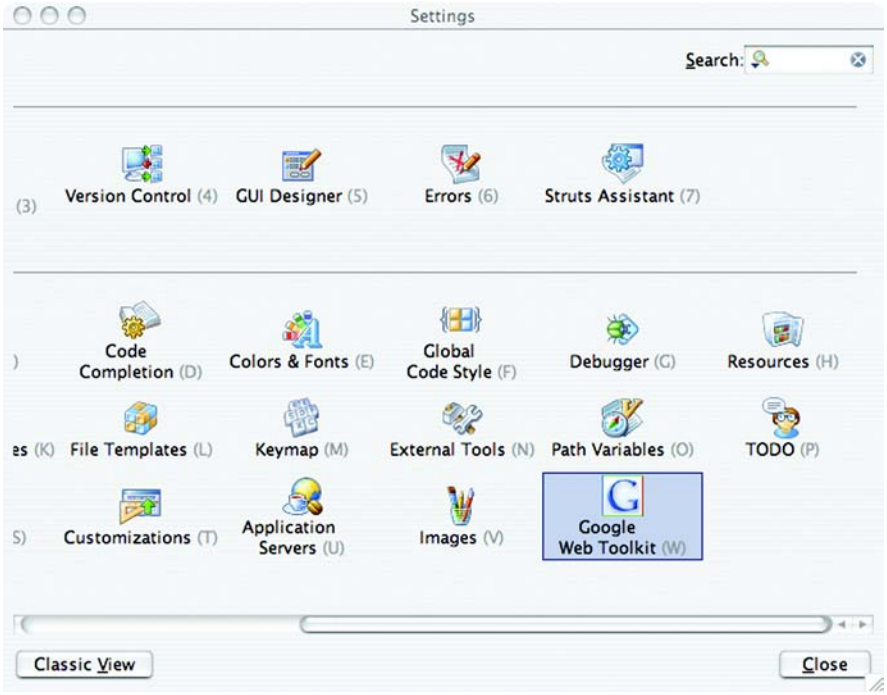

Figure 6.1 The Google Web Toolkit settings option in the IntelliJ IDEA settings panel. This setting can be used to specify your GWT Home folder.

in your module's gwt.xml file. This is an incredibly useful feature if you're dealing with multiple dependencies for packaged JSNI scripts, which we'll look at in this chapter. The second feature is the syntax highlighting, which is smart enough to skip into `/*-{ }-*/` blocks, giving you the advantages of JavaScript highlighting and formatting in your native methods, as pictured in figure 6.2. (Don't worry if the code in the figure seems a bit alien right now—we'll expand on this in section 6.3.)

The syntax highlighting feature alone makes IDEA a worthy candidate for your GWT development needs. Even better, if you're a well-established open source project, you can apply for a free license. Even at $250 for a personal edition, though, it's not out of reach for the average developer working on closed source projects, and a free thirty-day evaluation is available at http://www.jetbrains.com/.

Now that we have a development environment set up, it's time to get started. Our first task is to create a GWT module that encapsulates a JavaScript library. We'll start by wrapping the Moo.fx animation framework in a GWT module and build a small sample module that uses it. This will demonstrate the basics of working with JSNI, and it should better acquaint you with reading JSNI code.

Figure 6.2 IDEA's syntax highlighting of the JavaScript inside the `setDroppable()` method. The JavaScript syntax highlighting of JSNI methods is one of IDEA's greatest GWT features.

6.2 Wrapping JavaScript libraries

In the next two sections we're going to take a look at common patterns for integrating JavaScript and Ajax, and arm you with the tools and a general idea as to the process of using non-Java libraries with GWT. To this end, we'll generate wrappers for two JavaScript libraries so they can be used with GWT: Moo.fx and Script.aculo.us.

Moo.fx (http://moofx.mad4milk.net/) is one of our favorite JavaScript libraries. It's very small, and while it's limited in functionality, it can make a web application feel a lot more polished (maybe more polished than it actually is). The core Moo.fx library includes just three basic visual effects: width, height, and opacity. These let you change the respective attributes of a DOM object or toggle the presence of any object on screen. Also provided are several different transition types, including the sinoidal transition, which is a very natural and appealing looking transition that many applications you know and love, such as the Apple iLife suite, are already using.

We're going to use Moo.fx with a simple application that takes a photograph and alters its appearance. The example application, with no effects applied, is shown in figure 6.3.

The first step in creating our simple photo-effect application is to get the Moo.fx script ready for use in GWT. Because this is something we might want to reuse in other applications, we'll package the JSNI Moo.fx support as its own GWT module.

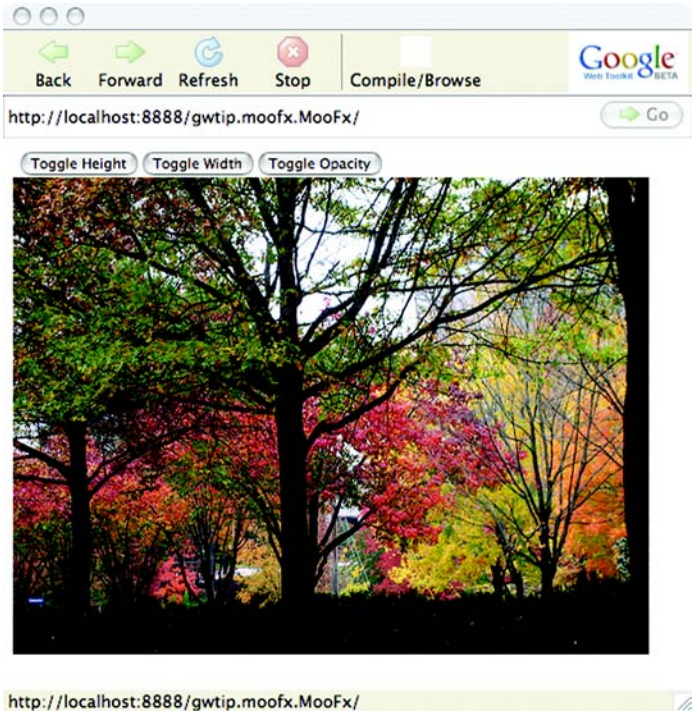

Figure 6.3
The sample Moo.fx application in hosted mode. We'll use Moo.fx to apply the three animated effects represented by the buttons above the photograph.

6.2.1 Creating a JavaScript module

As you saw earlier, the `<script>` tag makes directly including a JavaScript file in a module relatively simple. For this project, we're going to take the Moo.fx library and make it a GWT module that can be used with any GWT project. We'll start by creating the module descriptor, as shown in listing 6.2.

Listing 6.2 MooFx.gwt.xml module descriptor

```
<module>
    <script src="prototype.lite.js"><![CDATA[
        // Confirm that prototype is loaded
        if ($wnd.Class) {
            return true;
        } else {
            return false;
        }]]></script>
    <script src="moo.fx.js"><![CDATA[
        // Confirm that prototype is loaded
        if ($wnd.fx) {
            return true;
        } else {
            return false;
        }]]></script>
</module>
```

Include Prototype Lite for Moo.fx

Defined by Prototype Lite

Include Moo.fx script

Check for fx object

Once the module descriptor is filled out, we need to copy the JavaScript files into the `com.manning.gwtip.javascript.moofx.public` package, so they will be included with the compiled application.

Now that we know the script will load, we can use the object referenced by `fx` in our Java code to animate the image. To do that, we'll write a series of wrapper classes.

6.2.2 Creating wrapper classes

Each of Moo.fx's effect classes take two arguments as a constructor: the DOM element you wish to modify; and an associative array of options including how much time the effect should take to execute, a `Transition` object that determines the step value, and a callback that's notified when the effect is complete. For our JavaScript wrapper classes, we'll invoke the constructor and hold onto the created effect at the instance level, replicating the way Moo.fx is used in a pure JavaScript environment.

PROBLEM

We want to wrap a hierarchy of JavaScript classes from within Java.

SOLUTION

The best way to wrap a hierarchy of JavaScript classes from within Java is to replicate the exposure you need from JavaScript in a matching Java hierarchy. Moo.fx has several types of `Effect`, so in Java we'll create an `Effect` class and subclass it appropriately. We'll start with the `Effect` parent class, as shown in listing 6.3.

Listing 6.3 The `Effect` parent class

```
public class Effect {

    protected JavaScriptObject nativeEffect;
    protected EffectListener handler;
    protected UIObject uiObject;
    protected long duration;
    protected JavaScriptObject transition;

    protected Effect(UIObject uiObject, EffectListener handler,
            long duration , JavaScriptObject transition) {
        super();
        this.uiObject = uiObject;
        this.handler = handler;
        this.duration = duration;
        this.transition = transition;
    }

    public void toggle() {
        toggle(this.nativeEffect);
    }

    private native void toggle(
        JavaScriptObject effect)/*-{
        effect.toggle();
     }-*/;

    public void hide() {
        hide(this.nativeEffect);
    }

    private native void hide(
        JavaScriptObject effect) /*-{
        effect.hide();
     }-*/;

    public void clearTimer() {
        clearTimer(this.nativeEffect);
    }

    private native void clearTimer(
        JavaScriptObject effect) /*-{
        effect.clearTimer();
     }-*/;

    public void custom(int start, int end) {
        custom(this.nativeEffect, start, end);
    }

    private native void custom(JavaScriptObject nativeEffect,
         int start, int end) /*-{
        nativeEffect.custom(start, end);
    }-*/;
}
```

1 **Hold native implementation in JavaScriptObject**

Define callback interface for Java

Specify Transition to use

Wrap native methods

Define JSNI methods to invoke native code

Listing 6.4 shows where the `nativeEffect` comes from by presenting the `Height` class as a representative sample.

Listing 6.4 The `Height` Effect

```
public class Height extends Effect {

    public Height(UIObject uiObject, EffectListener handler,
            long duration, JavaScriptObject transition) {
        super(uiObject, handler, duration, transition);
        this.nativeEffect =
             getEffectClass(uiObject.getElement(),
                duration,
                handler,
            transition);
    }

    private native JavaScriptObject getEffectClass(
            JavaScriptObject element,
            long time,
            EffectListener handler,
            JavaScriptObject transitionFunction ) /*-{
        var complete = function(){
            if( handler != null ){
                handler.@gwtip.javascript.moofx.client.EffectListener::
                onComplete()();
            }
        };
        return new $wnd.fx.Height(element,
   { duration: time, transition: transitionFunction, onComplete::
                 complete });
    }-*/;
}
```

❷ **Call Effect constructor**

Create fx.Height object

Closure that wraps EffectListener callback

Construct fx.Height and return it

Now we have an `Effect` parent class and a `Height` child. This structure replicates the relationship the JavaScript objects have with each other.

DISCUSSION

The `Effect` class is mostly just a data structure, but it does contain the methods that invoke the JavaScript to perform an action. The native implementation in listing 6.3 ❶ will be set by the individual effect subclasses, such as the `Height` class. Mostly we're just capturing the `UIObject` that you want to perform the effect on and the other values that Moo.fx needs in that constructor.

The `Height` effect is a pretty simple class that simply passes the constructor arguments back up to `Effect` (❶ in listing 6.3) and then sets the `nativeEffect` with the proper implementation (❷ in listing 6.4). Here we're also extracting the DOM element enclosed by GWT's `UIObject` and passing it to the native object. Obviously, Moo.fx doesn't know anything about a GWT `UIObject`, but it does understand DOM elements. The `getElement()` method will become your friend when you're mixing GWT widgets and third-party JavaScript. Since the callback is handled by the `EffectListener` Java interface, we need to create a simple closure around it for the native effect to call when the transition is completed.

Speaking of transitions, Moo.fx includes four basic transition types that are stored as closures on the object referenced by the `fx` global variable. We'll expose these to

Java so they can be passed into the `Effect` subclasses using the `Transition` class, which is shown in listing 6.5. This class would be a standard example of a Java class holding constant values, except that we must have the native methods to retrieve the object references.

Listing 6.5 The `Transition` class

```
public class Transition {
    public static final JavaScriptObject SINOIDAL = getSinoidal();
    public static final JavaScriptObject LINEAR = getLinear();
    public static final JavaScriptObject CUBIC = getCubic();
    public static final JavaScriptObject CIRC = getCirc();

    private static native JavaScriptObject getSinoidal()
    /*-{return $wnd.fx.sinoidal;}-*/;
    private static native JavaScriptObject getLinear()
    /*-{return $wnd.fx.linear;}-*/;
    private static native JavaScriptObject getCubic()
    /*-{return $wnd.fx.cubic;}-*/;
    private static native JavaScriptObject getCirc()
    /*-{return $wnd.fx.circ;}-*/;
}
```

Our Moo.fx GWT wrapper is now ready to be compiled and packaged as a module for use in another GWT application.

6.2.3 Using the wrapped packages

To use the JSNI-wrapped module we have just created, we need to start by inheriting it in a different GWT application. This is old hat by now, but let's look at the module descriptor.

PROBLEM

We need to inherit a GWT JSNI-wrapped module in your GWT application.

SOLUTION

To use the JSNI wrapper module in the code, we need to inherit the module in the `gwt.xml` file as we would a Java module.

```
<module>
    <!-- Inherit the core Web Toolkit stuff. -->
    <inherits name='com.google.gwt.user.User'/>
      <inherits name='com.manning.gwtip.javascript.moofx.MooFx'/>
    <!-- Specify the app entry point class. -->
    <entry-point class='gwtip.moofx.client.MyEntryPoint'/>
</module>
```

Once the `<inherits>` elements are processed, GWT will extract the `moofx.public` package into the web application and will process the `<script>` elements from that module as they are encountered. We have packaged Prototype Lite with our Moo.fx module here, but it might also be wise to separate it out. If we were mixing several different JSNI modules that needed Prototype, or Prototype Lite, we wouldn't want the scripts to fight over which version was running at a particular time.

Next, we'll create a simple `EntryPoint` implementation to demonstrate each of the effects in listing 6.6.

Listing 6.6 An `EntryPoint` using the effects

```
public class MyEntryPoint implements EntryPoint {
    Image img = new Image(GWT.getModuleBaseURL()+"/fireworks.jpg");

    Height height = null;
    Width width = null;
    Opacity opacity = null;
    EffectListener listener = new EffectListener() {
        public void onComplete() {
            Window.alert("complete");
        }
    };
    public MyEntryPoint() {
        super();
    }

    public void onModuleLoad() {
        Button b = new Button("Toggle Height", new ClickListener() {

            public void onClick(Widget widget) {
                height = (height == null) ?
                    new Height(img,
                        listener,
                        500,
                        Transition.SINOIDAL
                        ) : height;
                height.toggle();
            }
        });
        RootPanel.get().add(b);

        b = new Button("Toggle Width", new ClickListener() {

            public void onClick(Widget widget) {
                width = (width == null) ? new Width(img,
                        listener,
                        500,
                        Transition.LINEAR
                        ) : width;
                width.toggle();
            }
        });
        RootPanel.get().add(b);
        b = new Button("Toggle Opacity", new ClickListener() {

            public void onClick(Widget widget) {
                opacity = (opacity == null) ?  new Opacity(img,
                        listener,
                        500,
                        Transition.CIRC
                        ) : opacity;
                opacity.toggle();
            }
```

1 **Hold effects at class level**

Create simple EffectListener

Create effect if it doesn't exist

```
        });
        RootPanel.get().add(b);

        RootPanel.get().add(img);
    }
}
```

Add image to panel

The `EntryPoint` example in listing 6.6 makes use of the wrapper classes we created for Moo.fx to animate a photograph with the three `Effect` types and binds three buttons to toggle each of them.

DISCUSSION

Most of this is fairly standard code for creating a basic set of widgets. The big thing you'll notice is that the `Effect` objects are held at the class level ❶, not created inside the `ClickListener`s for the buttons. This is because the `toggle()` method stores the original state of the DOM element in the native instance, so if you want `toggle()` to work, you need to keep the `Effect` reference around to restore the element to its original state. In your applications, you might think about creating your own `UIObject` subclasses that contain the appropriate effect and expose the `toggle()` or `clearTimer()` methods as needed. Finally, we simply call the toggle methods to make our image dance and sing, or maybe just dance. Figure 6.4 shows the application during an `Opacity` transition.

So far, you have seen Moo.fx used to load a JavaScript dependency into a GWT module, and you saw the basics of wrapping classes and using the GWT-created DOM

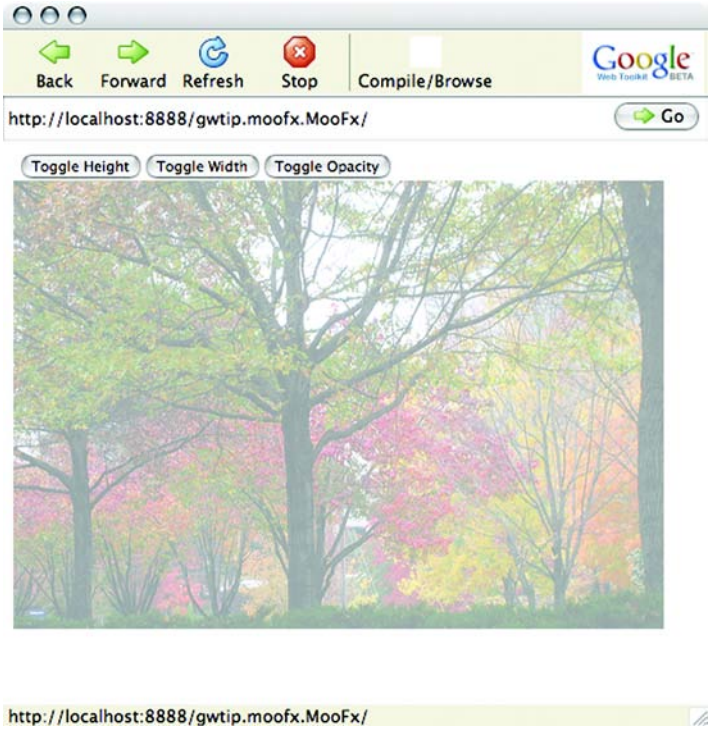

Figure 6.4 The image partially faded with the `Opacity` effect from Moo.fx. After you click the Toggle Opacity button, the image will fade out of existence.

elements in the JavaScript libraries. In the next section, we'll look in more detail at maintaining interactions between GWT components and JavaScript libraries. This is a pattern you'll find useful to repeat when building JSNI wrappers for many JavaScript libraries.

6.3 Managing GWT-JavaScript interaction

Script.aculo.us (http://script.aculo.us) is a more general-purpose UI library than Moo.fx. It includes an effects library very similar to the one provided by Moo.fx, and on top of that it has a `Slider` widget, an `Autocompleter` widget, an animation framework, a set of DOM utilities, and a testing framework. In the GWT world, a lot of this is uninteresting. Script.aculo.us's drag-and-drop functionality, on the other hand, is very interesting. While a native drag-and-drop system is on the drawing board for GWT, none is included right now, and this stands out as a major shortcoming when compared to other RIA toolkits.

In this sample project we're going to wrap the Script.aculo.us dragdrop.js support to enable our GWT application to use this feature. Note that we need to add prototype.js, effects.js, and dragdrop.js to the module definition, as these are requirements for Script.aculo.us.

Let's get to the meaningful classes. We'll start by implementing basic drag support.

6.3.1 Maintaining lookups

The core of Script.aculo.us's dragdrop.js is the `Draggable` class. This is a wrapper class that combines a standard DOM element with the code to enable dragging. Once the `Draggable` is created, the element can be dragged until the `destroy()` method is called. We want to make this as easy as possible on the developer, so we'll make instrumentation with the drag code available as a single static method call, and keep the static references around for calls to `destroy()` within the API code.

We'll start by building a simple application that lets us drag our image around the window, as seen in figure 6.5.

We'll be building a `DragAndDrop` class over the course of this section, so these early code samples will not have the complete code in them that you'll see in the samples from the Manning website. Our code will end up in the same completed state, but we'll be adding to the samples as we go.

Our Script.aculo.us support will look very much like the wrapper classes we created for the Moo.fx library, but it's important to note the maintenance of lookups in the Java code to coordinate between Java and JavaScript. To move back and forth between DOM element references and GWT classes, you will need to maintain lookup structures. As mentioned in the previous section, this is a pattern you'll find useful in any JSNI wrapper you're creating.

PROBLEM

Our JavaScript library is DOM `Element`-based, but your GWT code depends on `UIObject`.

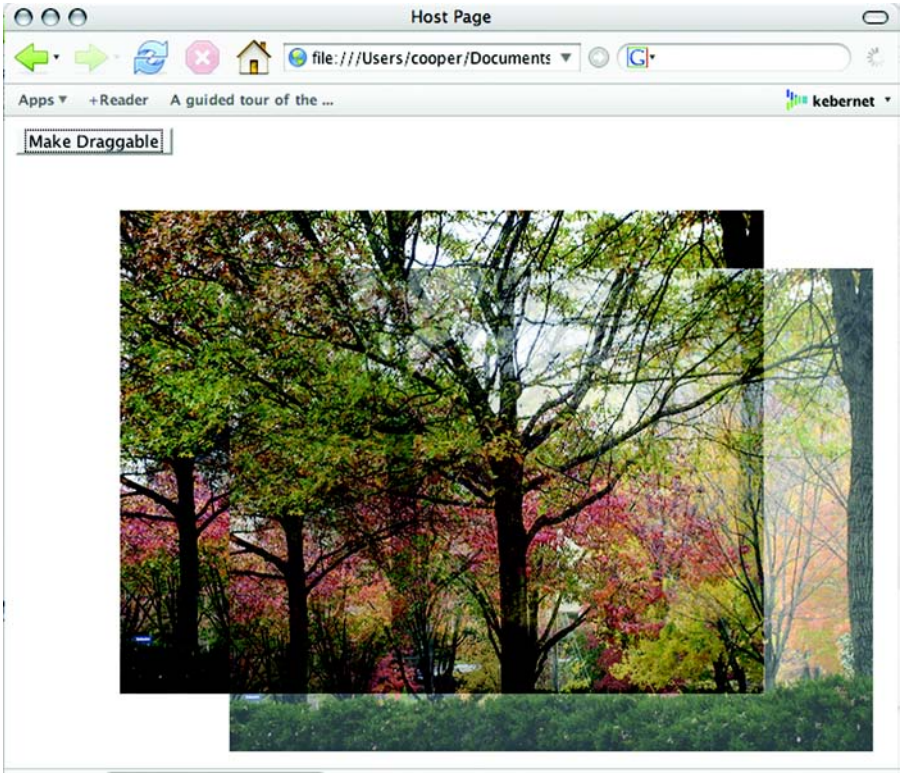

Figure 6.5 Dragging the image around in the window creates a ghost image as a placeholder. When the image is dragged, a partially transparent copy will follow the mouse.

SOLUTION

We'll start by creating the `setDraggable()` and `unsetDraggable()` methods and their supporting code, as shown in listing 6.7. This supporting code maintains lookups between `Elements` and `UIObjects`.

Listing 6.7 The first portion of the `DragAndDrop` class

```
public class DragAndDrop {

    public static final String
        CONSTRAINT_VERTICAL="vertical";
    public static final String
        CONSTRAINT_HORIZONTAL="horizontal";

    private static Map draggables = new HashMap();

    public static void setDraggable(UIObject source,
            UIObject handle,
            boolean revert,
            int zIndex,
            String constraint,
            boolean ghosting
```

Set fixed values for constraint

Map UIObject to JavaScriptObject (Draggable) ❶

❷ **Set options for Draggable**

```
            ) {
        if (draggables.get(source) != null) {
            unsetDraggable(source);
        }
        JavaScriptObject draggable = setDraggable(source.getElement(),
                ((handle == null) ? null : handle.getElement()),
                revert,
                zIndex,
                constraint,
                ghosting);
        draggables.put(source, draggable);
    }

    private static native JavaScriptObject setDraggable(
            JavaScriptObject element,
            JavaScriptObject handle,
            boolean revert,
            int zIndex,
            String constraint,
            boolean ghosting
            )/*-{
        return new $wnd.Draggable(
                element,
                {
                handle : handle,
                revert : revert,
                zindex : zIndex,
                constraint: constraint,
                ghosting: ghosting
                });
    }-*/;

    public static void unsetDraggable(
        UIObject source) {
        if (draggables.get(source) != null) {
            JavaScriptObject draggable =
                   (JavaScriptObject) draggables.remove(source);
            destroyDraggable(draggable);
        }
    }

    private static native void destroyDraggable(
        JavaScriptObject draggable)
    /*-{ draggable.destroy(); }-*/;
}
```

Unset current drag profile

Save JavaScriptObject to map

❸ **Remove drag profile from UIObject**

This class exposes several static methods to provide `UIObject`s with drag support and to keep track of the `JavaScriptObject`s created by Script.aculo.us.

DISCUSSION

It may seem that there are a lot of arguments in the methods at the beginning of the `DragAndDrop` class in listing 6.7 ❷, but they're mostly straightforward. Passing a `UIObject` to `setDraggable()` makes the `UIObject` draggable. Passing it to `unsetDraggable()` disables dragging. Each of the arguments is explained in table 6.2.

Table 6.2 Arguments for `setDraggable()`. These are part of the associative array in JavaScript but are fixed arguments in the Java methods.

Argument	Description
`source`	Specifies the `UIObject` we want to drag.
`handle`	Specifies the `UIObject` we want to be a handle, or the clickable part, for the draggable object. Think of the title bar on a window as a handle.
`revert`	Specifies whether the object should revert to its original position after the drag is completed.
`zIndex`	Specifies the CSS `z-index` property used to bring the draggable to the front.
`constraint`	Limits the dragging to one axis. Used with the static final `String` values, or `null`.
`ghosting`	Causes the dragging to use a partial transparency while leaving a duplicate of the element in place.

When we create a draggable, we store the resulting native object in the `draggables` lookup map ❶ so we can get back to it to call the `destroy()` method when necessary ❸. This is an important pattern to remember when integrating with JavaScript libraries: you should maintain the associations between GWT objects and the JavaScript objects on your own. This puts more of the onus on your code, sometimes even duplicating state that might exist in JavaScript. In the end, though, this will minimize unexpected errors caused by JavaScript incompatibilities or bugs.

In listing 6.8 we create a simple `EntryPoint` for our draggable application using `DragAndDrop`.

Listing 6.8 An `EntryPoint` using the `DragAndDrop` class

```
public class MyEntryPoint implements EntryPoint {
    Image img = new Image(GWT.getModuleBaseURL()+"/fireworks.jpg");
    VerticalPanel panel = new VerticalPanel();

    public MyEntryPoint() {
        super();
    }

    public void onModuleLoad() {
        Button b = new Button("Make Draggable", new ClickListener() {
            public void onClick(Widget widget) {
                DragAndDrop.setDraggable(img,
                      null, false, 1000, null, true);
            }
        });
        panel.add(b);

        panel.add(img);
        RootPanel.get().add(panel);
    }
}
```

Set up options

In listing 6.8 we make the image draggable with no handle, no reverting to the original position, a z-index of 1000, no constraints, and ghosting enabled. You've already seen the results of this in the ghosted image of the trees (figure 6.5).

This approach is pretty easy, and it works well, but right now it's simply browser candy. If we want to make dragging useful, we need to know when it happens. We need to add some event listeners.

6.3.2 Daisy-chaining Java listeners into JavaScript closures

In its native operation, Script.aculo.us supports capturing events in two ways, and each of these is handled using different metaphors. First, there is a `change()` closure that can be attached to the `Draggable` object. This is fired whenever something happens to the drag-enabled DOM element. The other is through an Observer pattern object, much like a standard Java event listener. These listeners implement `onStart()`, `onDrag()`, and `onEnd()`, receiving an event name and the `Draggable` object that fired the event as arguments. These are a bit limiting, so we'll wrap them in something that's a little more robust. We will look at each of these methods in turn.

PROBLEM

JavaScript provides a simple closure for event notifications, but our Java code needs an Observer pattern to receive events.

SOLUTION

First, we need the `DragListener` interface as shown in listing 6.9.

Listing 6.9 `DragListener`

```
public interface DragListener {
    void onChange(UIObject source);
}
```

There is not much to the `DragListener` interface. To get from the JavaScript event to the `UIObject`, we custom-generate the closure that gets called, and store the `UIObject` in the local scope.

Next we want to track the listeners in the `DragAndDrop` class. In listing 6.10 we continue to add on to the `DragAndDrop` class to add this support.

Listing 6.10 Changing `DragAndDrop` to support `DragListener`

```
private static Map dragListeners = new HashMap();

public static void addDragListener(UIObject source,
         DragListener listener) {
    JavaScriptObject draggable =
          (JavaScriptObject) draggables.get(source);
    if (draggable == null) {
        throw new RuntimeException("That is not a draggable object.");
    }
    List listeners = (List) dragListeners.get(source);
    if (listeners == null) {
        listeners = new ArrayList();
```

Lookup of UIObject to DragListeners

Validate passed draggable object

```
        dragListeners.put(source, listeners);
    }
    listeners.add(listener);
    unsetChange(draggable);
    for (Iterator it = listeners.iterator(); it.hasNext();) {
        appendListener(source, draggable,
            (DragListener) it.next() );
    }
}

private static native void unsetChange(JavaScriptObject draggable)
/*-{draggable.options.change = null;}-*/;

private static native void appendListener(
        UIObject source, JavaScriptObject draggable,
        DragListener listener )/*-{
    var oldChange = draggable.options.change;
    var newChange = function(element) {
        if (oldChange) oldChange(element);
        listener.@gwtip.javascript.scriptaculous.client.DragListener::
        onChange(Lcom/google/gwt/user/client/ui/UIObject;)(source);
    }
    draggable.options.change = newChange;
}-*/;

public static void remoteDragListener(
    UIObject source, DragListener listener) {
    JavaScriptObject draggable =
       (JavaScriptObject) draggables.get(source);
    if (draggable == null) {
        throw new RuntimeException("That is not a draggable object.");
    }
    List listeners = (List) dragListeners.get(source);
    if (listeners != null) {
        listeners.remove(listener);
        unsetChange(draggable);
        for (Iterator it = listeners.iterator(); it.hasNext();) {
            appendListener(source, draggable, (DragListener) it.next());
        }
    }
}
```

❶ **Add listener to List**

Remove current options.change closure

New Listener calls into options.change closure

❷ **Daisy-chain closures**

❸ **Set new closure**

❹ **Remove listener and rebuild**

Here we keep a list of Java listeners around and rebuild a chain of JavaScript closures to fire them as part of the single closure the Script.aculo.us library provides for event notification.

Discussion

Listing 6.10 might, at first glance, seem very complicated, but it's actually not that bad. First, we keep an `ArrayList` of listeners keyed to `UIObjects`. When it comes time to add or remove a listener, we modify the `ArrayList` as needed ❶ ❹. Then we cycle through the listeners and add them to a daisy chain of closures on the `options.change` property of the `draggable` element ❷ ❸.

This daisy-chaining ❷ might seem odd if you have not done Ajax work in the past, but this technique allows a single closure object to support multiple methods. This is a

common idiom for letting scripts overload the <body> element's onload event. Each time we want to add a closure, we take the current one and hold it. Then, in the next closure method, we first call the original, if needed, and execute the current event. When the final closure is called, we climb up to the top of the chain and execute each closure in turn. In this case, each of them fires the onChange event of the listener they were created to handle.

Now that we're supporting DragListeners, we can modify the EntryPoint class to show that they are working. Listing 6.11 shows the new listeners added to our draggable.

Listing 6.11 MyEntryPoint.onModuleLoad handling listeners

```
public void onModuleLoad() {
    Button b = new Button("Make Draggable", new ClickListener(){

        public void onClick(Widget widget) {
            DragAndDrop.setDraggable(img, null, false, 1000, null, true);
            DragAndDrop.addDragListener(img, new DragListener() {
                public void onChange(UIObject source) {
                    panel.add(new Label("Listener 1 event."));
                }
            });
            DragAndDrop.addDragListener(img, new DragListener() {
                public void onChange(UIObject source){
                    panel.add(new Label("Listener 2 event."));
                }
            });
        }
    });
    panel.add(b);
    panel.add(img);
    RootPanel.get().add(panel);
}
```

Now when we drag the image around the window, we'll see Labels that indicate events filling up the bottom of the screen, as shown in figure 6.6.

Because our daisy-chained closures call the previous closure first, they execute in the order they were added. If, for instance, we wanted to allow one listener to consume the event and stop processing, we could have the DragListener interface return a boolean indicating whether processing should continue. If any closure gets a false returned from its predecessor, it simply returns false and exits.

This is pretty useful functionality, but we might want to do more with it than we currently are. Most importantly, these events don't tell us what kind of event fired them, meaning we don't know the difference between picking up, moving, and dropping. What we're lacking is the global Observer support.

6.3.3 Maintaining listeners in Java

The DragObserver is both powerful and complex to implement. We have already discussed the three methods Script.aculo.us supports—onStart(), onDrag(), and onEnd()—so let's just start with the interface.

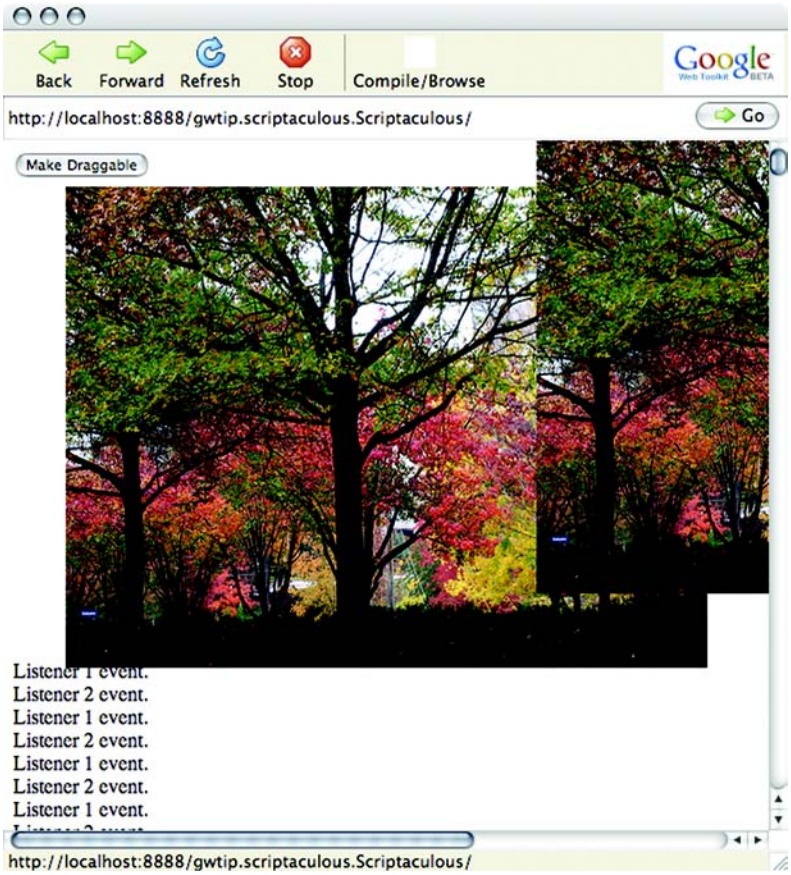

Figure 6.6 `DragListener` events rendering to the screen as the image is dragged. The text at the bottom of the page shows the drag events firing as the image is dragged off to the right.

PROBLEM

The JavaScript Observer pattern is not addressable from our GWT Java code.

SOLUTION

To proxy between JavaScript and Java observers, we must start by defining the interface we will use to capture events, as shown in listing 6.12.

Listing 6.12 The `DragObserver` Interface

```
public interface DragObserver {
    void onStart(String eventName, UIObject source);
    void onDrag(String eventName, UIObject source);
    void onEnd(String eventName, UIObject source);
}
```

You'll notice here that we're firing the events with `UIObject`. The JavaScript native `Draggables` object is actually going to pass in the `Draggable` native implementation, which we're storing in a `HashMap` already. However, we'll create a new `HashMap` of `Elements` to `UIObjects` to make this lookup easier to use when we get into drop targets. Let's look at how we can wire up the `DragAndDrop` class to support the drop observers. Listing 6.13 represents additions to the class.

Listing 6.13 Changes to the `DragAndDrop` class to support drop observers

```
private static Map elementsToUIObjects = new HashMap();
private static List dragObservers = new ArrayList();

static{
    initDragObservers();
}

private static native void initDragObservers()/*-{
    var observer = {
            onStart: function(name, draggable, event) {
                @gwtip.javascript.scriptaculous.client.DragAndDrop::
fireOnStart(Ljava/lang/String;Lcom/google/gwt/user/client/Element;)(
                    name, draggable.element);
            },
            onEnd: function(name, draggable, event) {
                @gwtip.javascript.scriptaculous.client.DragAndDrop::
fireOnEnd(Ljava/lang/String;Lcom/google/gwt/user/client/Element;)(
                  name, draggable.element);
            },
            onDrag: function(name, draggable, event) {
                @gwtip.javascript.scriptaculous.client.DragAndDrop::
               fireOnDrag
      (Ljava/lang/String;Lcom/google/gwt/user/client/Element;)
      (name, draggable.element);
            }
    };

    $wnd.Draggables.addObserver(observer);
}-*/;

private static void fireOnStart(String name,
    Element element) {
    for (Iterator it = dragObservers.iterator();
        it.hasNext();
       ) {
        ((DragObserver) it.next())
                .onStart(
                    name,
                    (UIObject) elementsToUIObjects
                        .get(element)
                );
    }
}
private static void fireOnDrag(String name, Element element) {
    for (Iterator it = dragObservers.iterator(); it.hasNext();) {
```

1 **Create data structures**

2 **Fire listeners from a single observer**

Pass element, not draggable

Add observer to Draggables

Look up UIObject by element

3 **Fire events to registered observers**

```
        ((DragObserver) it.next())
                .onDrag(
                    name,
                    (UIObject) elementsToUIObjects.get(element)
                );
    }
}
private static void fireOnEnd(String name, Element element) {
    for (Iterator it = dragObservers.iterator(); it.hasNext();) {
        ((DragObserver) it.next())
                .onEnd(
                    name,
                     (UIObject) elementsToUIObjects.get(element)
                );
    }
}

public static void addDragObserver(DragObserver observer) {
    dragObservers.add(observer);
}
public static void removeDragObserver(DragObserver observer) {
    dragObservers.remove(observer);
}
```

Look up UIObject by element

Now we have the Java interface for `DragObserver` and methods for adding observers to and removing them from the `DragAndDrop` class.

Discussion

What we're actually doing in listing 6.13 is creating the list of `Observers` in Java ❶, registering a single JavaScript `Observer` that calls back into our Java ❷, and then firing the Observers as appropriate ❸.

An astute reader might notice something missing here. For brevity, we omitted adding the `elementsToUIObjects.remove()` call in the `unsetDraggable()` method and the `elementsToUIObjects.put()` call in the `setDraggable()` method. These, however, need to be there to make sure the lookup is populated. Next we simply add a few lines to the entry point so we can see these events firing:

```
DragAndDrop.addDragObserver(new DragObserver(){
    public void onStart(String eventName, UIObject source) {
        panel.add(new Label(
            eventName +" "+ ((Image) source).getUrl()));
    }

    public void onEnd(String eventName, UIObject source) {
        panel.add(new Label(
            eventName +" "+ ((Image) source).getUrl()));
    }

    public void onDrag(String eventName, UIObject source) {
        panel.add(new Label(
           eventName +" "+ ((Image) source).getUrl()));
    }

});
```

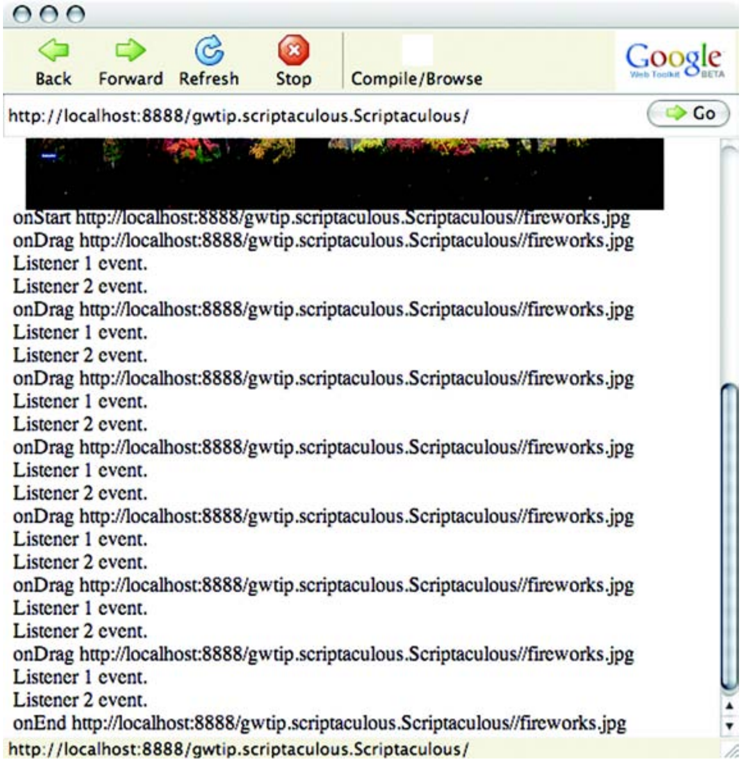

Figure 6.7 `DragListeners` and `DragObservers` fired through a drag. Here we see the `DragObserver` events firing, followed by the `DragEvents` we saw in the previous section. Notice that the observers see `onStart()` and `onDrag()`, and not just the basic event.

Now when we run the sample app, as shown in figure 6.7, we can see the order in which the events are fired during a short drag operation.

In these last two sections, you have seen two different techniques for translating JavaScript events into Java. The *right* way varies by situation and the code you're working with. Sometimes it's easier to work with the JavaScript listeners or to daisy-chain closures on an object. Sometimes it's easier to simply move the event-firing logic into Java and work from there. The big, and rather blindingly obvious, lesson here is to look at your JavaScript and think about what you need, then get into Java and select the best approach from there.

We aren't quite done yet, though. We still need drop support, and that means a whole new set of Java to JavaScript communications.

6.3.4 Conversion between Java and JavaScript

Drop targets are created in Script.aculo.us by calling `Droppables.add()`. We're going to wrap this in a Java method, and then we're going to support `DropListeners` within

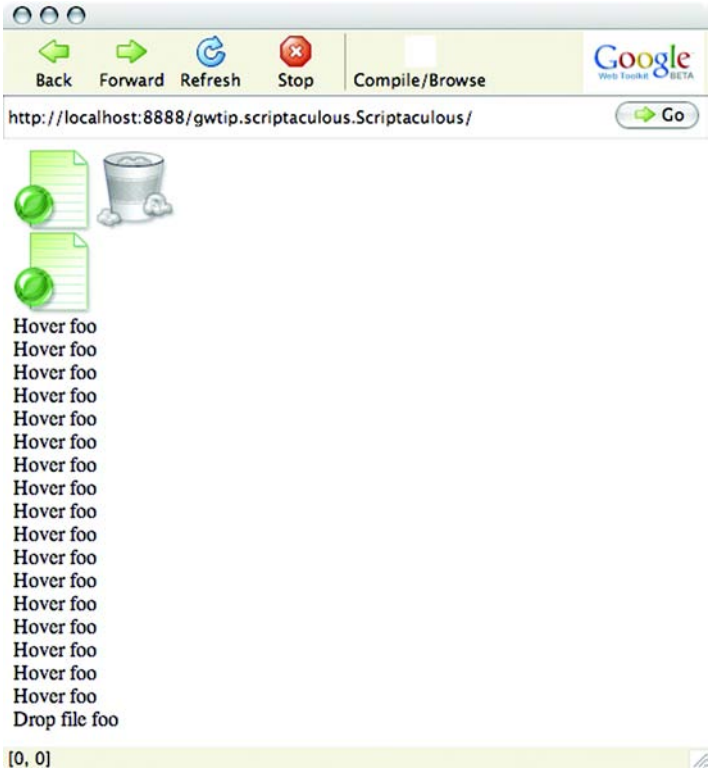

Figure 6.8 Removing one of three file icons by dragging it to the trash and dropping it. Notice the hover events followed by the drop event as a file icon is dragged and dropped onto the trash can. (Icons courtesy of kellishaver.com.)

the `DragAndDrop` class. Finally, we'll create a simple `EntryPoint` to show the drop operations. In the end, we'll be able to drag and drop elements in the GWT context, as shown in figure 6.8.

You have seen two different methods for dealing with events and keeping track of lookups to smooth interactions between Java and JavaScript. In this section we'll combine these two to handle drop events. We'll also elaborate on an important and common troublesome task in working with JSNI—dealing with arrays.

PROBLEM

We need to deal with complex data conversions between Java and JavaScript, such as array conversions.

SOLUTION

First let's look at the code we need to add to the `DragAndDrop` class to register a droppable object. The `setDroppable()` method takes the arguments outlined in table 6.3.

You'll notice that `acceptClasses` and `containment` are both arrays. You'll recall from the first part of this chapter that arrays are completely opaque to your JavaScript code, so we'll have to convert between Java arrays and JavaScript arrays. Listing 6.14 shows this and the other modifications needed for our `DragAndDrop` support class to handle drop targets.

Table 6.3 Arguments for `setDroppable()`. As with `setDraggable()`, many of these are encapsulated in the JavaScript associative array. In Java we use named arguments.

Argument	Description
`target`	The `UIObject` to enable as a drop target.
`acceptClasses`	A `String` array of `Style` class values specifying what can be dropped on the target.
`containment`	A `UIObject` array of which dropped items must be children.
`hoverClass`	A `Style` class that will be applied to the target when something is hovering over it but not dropped.
`overlap`	One of `OVERLAP_VERTICAL`, `OVERLAP_HORIZONTAL`, or `null`. If set, the drop target will only react to a dragged item if it's overlapping by more than 50% on the given axis.
`greedy`	A flag indicating that the drop target should end calls to targets that may be beneath it once its own events have fired.

Listing 6.14 `DragAndDrop` additions to handle drop targets

```
public static final String OVERLAP_VERTICAL="vertical";
public static final String OVERLAP_HORIZONTAL="horizontal";

private static Set droppables = new HashSet();
private static Map dropListeners = new HashMap();

public static void setDroppable(UIObject target,
        String[] acceptClasses,
        UIObject[] containment,
        String hoverClass,
        String overlap,
        boolean greedy
        ){
    JavaScriptObject acceptNative =
        javaArrayToJavaScriptArray(  acceptClasses);
    JavaScriptObject containmentNative = null;
    if (containment != null) {
        Element[] containElements = new Element[containment.length];
        for (int i = 0; i < containment.length; i++){
            containElements[i] = containment[i].getElement();
        }
        containmentNative = javaArrayToJavaScriptArray(containElements);
    }
    droppables.add(target);
    setDroppable(target.getElement(),
            acceptNative,
            containmentNative,
            hoverClass,
            overlap,
            greedy,
            target);
}
```

Hold droppables and listeners

Pass arguments from table 6.3

Convert arrays

```
private static JavaScriptObject
    javaArrayToJavaScriptArray(Object[] array) {
    JavaScriptObject jsArray = null;
    for (int i = 0; array != null && i < array.length; i++) {
        if (array[i] instanceof String) {
            jsArray = addToArray(jsArray, (String) array[i]);
        } else {
            jsArray = addToArray(jsArray, (JavaScriptObject) array[i]);
        }
    }
    return jsArray;
}
```

Switch on object type

```
private static native JavaScriptObject addToArray(
    JavaScriptObject array, JavaScriptObject object)
/*-{
    if (array == null) array = new Array();
    array.push(object);
}-*/;
private static native JavaScriptObject addToArray(
        JavaScriptObject array, String object)/*-{
    if (array == null) array = new Array();
    array.push(object);
}-*/;
```

Create arrays and push each element

```
private static native void setDroppable(Element target,
        JavaScriptObject acceptClasses,
        JavaScriptObject containment,
        String hoverClass,
        String overlap,
        boolean greedy,
        UIObject uiObject)/*-{
    $wnd.Droppables.add(target,
        {
           accept: acceptClasses,
           containment: containment,
           hoverclass: hoverClass,
           overlap: overlap,
           greedy: greedy,
           onDrop: function(draggable,
                       droppable, overlap) {
              @gwtip.javascript.scriptaculous.client.DragAndDrop::
              fireOnDrop
             (Lcom/google/gwt/user/client/ui/UIObject;
              Lcom/google/gwt/user/client/Element;)
             (uiObject, draggable );
                    },
           onHover: function(draggable, droppable) {
              @gwtip.javascript.scriptaculous.client.DragAndDrop::
              fireOnHover
             (Lcom/google/gwt/user/client/ui/UIObject;
              Lcom/google/gwt/user/client/Element;)
             (uiObject, draggable );
                    },
        });
}-*/;
```

Create onDrop event closure

Create onHover event closure

We have added the methods to create drop targets and added utility methods to convert between the Java and JavaScript array types.

DISCUSSION

The code we used in listing 6.14 to add drop support should be looking familiar to you at this point. The big change here is that we have to convert the Java arrays into JavaScript arrays, enclosed by `JavaScriptObject`s, to pass them around. This convoluted structure is necessary because you can't add to the JavaScript array from Java, nor can you read from a Java array within JavaScript.

The final portion of the `DragAndDrop` class, shown in listing 6.15, involves setting up the event registry and removing droppables. This is not unlike the `DragObserver` support we added earlier.

Listing 6.15 Finishing up the `DragAndDrop` class

```
public static void unsetDroppable(UIObject target) {
    if (droppables.contains(target)) {
        unsetDroppable(target.getElement());
        dropListeners.remove(target);
        if (draggables.get(target) != null) {
            elementsToUIObjects.remove(target);
        }
    }
}

private static native void unsetDroppable(Element element)
/*-{ $wnd.Droppables.remove(element ); }-*/;

private static void fireOnHover(
  UIObject source, Element hovered) {
    UIObject hoveredUIObject =
        (UIObject) elementsToUIObjects.get(hovered);
    List listeners = (List) dropListeners.get(source);
    for (int i = 0; listeners != null && i < listeners.size(); i++) {
        ((DropListener) listeners.get(i))
              .onHover(source, hoveredUIObject);
    }
}

private static void fireOnDrop(
   UIObject source, Element dropped) {
    UIObject droppedUIObject =
        (UIObject) elementsToUIObjects.get(dropped);
    List listeners = (List) dropListeners.get(source);
    for (int i = 0; listeners != null && i < listeners.size(); i++) {
        ((DropListener) listeners.get(i))
               .onDrop(source, droppedUIObject);
    }

}
public static void addDropListener(
  UIObject target, DropListener listener) {
    ArrayList listeners = (ArrayList) dropListeners.get(target);
    if (listeners == null) {
```

Clean up listeners

Ensure drag features don't collide

Convert back to UIObject

```
            listeners = new ArrayList();
            dropListeners.put(target, listeners);
        }
        listeners.add(listener);
    }
```

You'll notice in listing 6.15 that we're checking the draggables before we remove them from the `elementsToUIObjects` lookup hash. You need to add this to the `unsetDraggable()` method to prevent collisions. Now we have a fully functional `DragAndDrop` class.

NOTE If you're using this library in your own application, remember that you *must* remove droppables before taking them off the screen. If you don't, the entire Script.aculo.us dragdrop library will break down.

The final step in bringing our little example to fruition is writing the entry point. This is shown in listing 6.16, and it demonstrates how a drag-and-drop system that instruments classes in place, rather than requiring the developer to implement special drag-and-drop enabled classes, is easy to use. If you have worked with a very complex DnD system like Java Swing, the simplicity of using this kind of system is obvious.

Listing 6.16 `EntryPoint` for `DragAndDrop`

```
public class MyEntryPoint implements EntryPoint {
    VerticalPanel panel = new VerticalPanel();
    HorizontalPanel hpanel = new HorizontalPanel();
    VerticalPanel output = new VerticalPanel();

    public MyEntryPoint() {
        super();
    }

    public void onModuleLoad() {
        Image trash = new Image("trash.png");
        String[] accept = {"file"};
        DragAndDrop.setDroppable(trash, accept, null, null, null, true);
        DragAndDrop.addDropListener(trash, new DropListener() {
            public void onHover(UIObject source, UIObject hovered) {
                output.add(new Label("Hover "+((File) hovered).name));
            }

            public void onDrop(UIObject source, UIObject dropped) {
                output.add(new Label("Drop file "+
                  ((File) dropped).name));
                panel.remove((File) dropped);
            }
        });

        hpanel.add(panel);
        hpanel.add(trash);
        File f = new File("foo", "webpage.png");
        f.setStyleName("file");
        panel.add(f);
        DragAndDrop.setDraggable(f, null, true, 1000, null, true);
```

Create Panel for Event output

Specify name and image for Files

Create three files as draggable

```
            f = new File("bar", "webpage.png");
            f.setStyleName("file");
            panel.add(f);
            DragAndDrop.setDraggable(f, null, true, 1000, null, true);

            f = new File("baz", "webpage.png");
            f.setStyleName("file");
            panel.add(f);
            DragAndDrop.setDraggable(f, null, true, 1000, null, true);

            RootPanel.get().add(hpanel);
            RootPanel.get().add(output);
        }
    }
```

Create three files as draggable

Create three files as draggable

With listing 6.16 we have completed our basic example. Keep in mind that while we have demonstrated the key concepts, you would be better served moving the setup for your drag-and-drop operations out into your UI classes, and to manage events from your Controller level rather than in an entry point. However, this simple example keeps things concise and demonstrates the lifecycle of drop targets.

As you can see, there are lots of design issues to keep in mind when you're creating your JSNI wrappers. There is an easier way, for those who feel adventurous.

6.4 Wrapping JavaScript with GWT-API-Interop

While wrapping JavaScript APIs with GWT code is still the only officially sanctioned way to create JSNI modules, it can be slow going. Fortunately, there is an easier way. The GWT-API-Interop project (http://code.google.com/p/gwt-api-interop) provides a compile-time linking between GWT interfaces and JavaScript libraries.

We say that wrapping manually is the only *sanctioned* method because of this library's readme notes: "This project should be considered experimental, AS-IS, and generally unsupported at this time." While not officially supported, it's the basis of other code that the Google team has shipped for GWT. Your mileage may vary. We'll take a quick look at how to use this very handy module, in spite of this warning.

PROBLEM

Wrapping a large JavaScript library by hand is tedious.

SOLUTION

Using the GWT-API-Interop package can greatly simplify the process of wrapping a JavaScript library. This module uses a set of annotations on GWT-style Java interfaces to generate wrappers for JavaScript libraries at compile time.

The first step in the process is to import the JSIO module into your project. This will bring in the appropriate code generators:

```
<inherits name="com.google.gwt.jsio.JSIO"/>
```

Next, you need to create Java interfaces that map to your JavaScript library and extend `com.google.gwt.jsio.client.JSWrapper`. These will then be annotated with meta information about how they should be mapped. Listing 6.17 shows a simple interface mapped to a hypothetical JavaScript type.

Listing 6.17 Defining a JSIO-wrapped JavaScript object

```
/**
 *
 * @gwt.constructor $wnd.MyJavaScriptObject
 */
public interface MyJavaScriptObject extends JSWrapper {

    /**
     * @gwt.fieldName doMethod
     */
    void doMethod(String string, int integer);

    int intAttribute();
}
```

❶ Specify constructor

❷ Specify method field

❸ Get simple attribute

DISCUSSION

Here we're creating a `JSWrapper` interface and specifying two annotations. First, we define the constructor function from the JavaScript API ❶. This will be called when the interface is instantiated using `GWT.create()`. Next, we define a method and use the `fieldName` annotation to associate it with the JavaScript field containing the function that implements the method ❷. For simple value attributes, we can simply name a method based on the attribute of the JavaScript object we want to return ❸.

Wow, that's easy! There are some caveats here, though. First, we still have to deal with issues of arrays. For this, the JSIO module includes the `JSList` interface. This is an implementation of `java.util.List` that you can use as a wrapper for JavaScript `Array` return or argument types. Note that you must annotate uses of `JSList` with the `gwt.typeArgs` annotation the same way you would with RPC calls or `IsSerializable` objects.

Another issue we spent a good deal of time on earlier was dealing with callbacks and events from JavaScript libraries. The JSIO module includes a means of handling this as well. The `JSFunction` abstract class allows you to create callbacks and pass them in with method calls. Listing 6.18 shows a brief example of this.

Listing 6.18 Creating a callback class for a JSFunction

```
/**
 * @gwt.exported onMyEvent
 */
public abstract class MyCallback extends JSFunction {

    void onMyEvent(String callbackArgument);
}
```

❶ Specify JavaScript function to be called

❷ Specify method to call

Here we specify the method signature we want invoked in our GWT Java code ❷. To specify the callback from this method, we use the `exported` annotation ❶. This tells the code generator that the `onMyEvent()` Java method should be exposed, name intact, to JavaScript callers, and we're declaring this with a single function. You could, however, use the `exported` annotation on multiple methods if there are multiple possible states, such as `onSuccess` and `onFailure`. This will convert these methods into JavaScript closures that can be applied to asynchronous method calls, which works for

simple calls where you might pass in a callback to the method call. You can also use the `exported` annotation on a mixed abstract class and use one of the techniques discussed in the previous section to handle Observer pattern listeners, such as using the `exported` annotation at the method level on an abstract class that extends `JSWrapper`.

The JSIO module includes a number of other annotations. Table 6.4 provides a complete list.

Table 6.4 A complete list of JSIO-supported annotations

Annotation	Location	Description
`gwt.beanProperties`	Class, method	This annotation indicates that methods that look like bean-style property setters should be generated so as to read and write object properties rather than import functions. This is most useful with JSON-style objects. The setting may be applied on a per-method basis in an imported class and may be overridden on a per-method basis by `gwt.imported`. If the backing object does not contain data for a property accessor, `null`, `0`, `' '`, `false`, or an empty `com.google.gwt.jsio.client.JSList` will be returned.
`gwt.constructor`	Class, method	The annotation `gwt.constructor` may be applied to a class to specify a JavaScript function to execute when constructing a `JSWrapper` to use as the initial backing object. A JavaScript `Date` object could be created by using the value `$wnd.Date`. If the `gwt.constructor` annotation is applied to a method within a `JSWrapper` and the method is invoked, the parameters of the method will be passed to the named global function, and the resulting JavaScript object will be used as the backing object.
`gwt.exported`	Method	Individual Java functions may be exported to JavaScript callers by declaring a `gwt.exported` annotation on a concrete Java method within a `JSWrapper`. The Java method will be bound to a property on the backing object per the class's `NamePolicy` or a `gwt.fieldName` annotation on the method. When applied at the class level to a `com.google.gwt.jsio.client.JSFunction`, it specifies which of the methods declared within to export as a JavaScript `Function` object.
`gwt.fieldName`	Method	When implementing a bean property accessor, the default `NamePolicy` will be used unless a `gwt.fieldName` annotation appears on the property's getter or setter. This is also used with imported and exported functions to specify the object property to attach to.
`gwt.global`	Class	The annotation `gwt.global` is similar to `gwt.constructor`, however it may be applied only at the class level and the value is interpreted as a globally accessible object name rather than as a function.

Table 6.4 A complete list of JSIO-supported annotations *(continued)*

Annotation	Location	Description
`gwt.imported`	Method	This is an override for methods within classes annotated with `gwt.beanProperties`.
`gwt.namePolicy`	Class	This annotation specifies the default transformation to use when converting bean property accessor function names to fields on the underlying `JavaScriptObject`. The valid values for the `namePolicy` are the field names on the `com.google.gwt.jsio.rebind.NamePolicy` class, or the name of a class that implements `NamePolicy`.
`gwt.noIdentity`	Class	This annotation suppresses the addition of the `__gwtObject` property on the underlying `JavaScriptObject`. The object identity of the `JSWrapper` will no longer maintain a one-to-one correspondence with the underlying `JavaScriptObject`. Additionally, `com.google.gwt.jsio.client.JSWrapper#setJavaScriptObject` will no longer throw `com.google.gwt.jsio.client.MultipleWrapperException`.
`gwt.readOnly`	Class	This annotation prevents the generated `JSWrapper` implementation from containing any code that will modify the underlying `JavaScriptObject`. This implies `gwt.noIdentity`. Invoking a bean-style getter when the underlying `JavaScriptObject` does not contain a value for the property will result in undefined behavior.

While it's important to remember that the GWT-API-Interop/JSIO module is unsupported and experimental, it can save you a great deal of hand-coding if you wish to use legacy or third-party JavaScript. We have only taken a brief look at its use here, but you can find more documentation at the project website (http://code.google.com/p/gwt-api-interop).

6.5 Summary

The JSNI component is both the most powerful and the most fraught with danger in the Google Web Toolkit. While you can use it to add nearly unheard-of ease of use to your Java classes, as the drag-and-drop example demonstrates, moving data successfully between the two environments can be an exercise in frustration. Mysterious errors crop up, and unlike GWT's compiled Java, in JavaScript you have to deal with cross-browser compatibility on your own.

In this chapter you have seen most of the issues involved with JSNI integration and picked up a couple of useful GWT modules to boot. One of the great advantages of mixed-mode development with GWT is that you can maintain the structure and common typing of Java and—only when you need to—break down to the lower-level flexibility of JavaScript to manipulate the browser.

Packaging JSNI libraries as easily reusable GWT modules is a great way to ease the development of your application. But once you start using these techniques, you need to be well versed at packaging and deploying your GWT applications. While we have seen simple examples running in the GWT development environment thus far, we'll take a closer look at building and deploying GWT applications for production in the next chapter.

JAVA / WEB

GWT IN PRACTICE

Robert T. Cooper and Charlie E. Collins

Free ebook
SEE INSERT

With the Google Web Toolkit (GWT), you can build rich internet applications in Java using a Swing-like framework that provides dozens of pre-built components. GWT automatically converts your Java into JavaScript, so you can focus on application design and functionality without concern for browser quirks. This powerful tool opens up many new possibilities for web development, like using Java test and build tools, integrating with frameworks like the Java Persistence API, and easily reusing code throughout the layers of your application.

GWT in Practice is an example-driven, code-rich book written for web developers who have knowledge of the basics of GWT. You'll find scores of cookbook-style solutions for common and uncommon needs like drag-and-drop support for UI elements, data binding, processing streaming data, handling application state, automated builds, and continuous integration. And you will appreciate the attention it pays to the problem of integrating GWT with your existing applications and services.

What's Inside

- Create and customize widgets
- Learn the ins and outs of RPC
- Package, build, and test with Maven, Ant, and JUnit
- Work with the Java Persistence API
- Use GWT from Eclipse, NetBeans, and IDEA

Robert Cooper and **Charlie Collins** are Atlanta-based Java EE developers. Both contribute to many open source projects, including the gwt-maven plugins that support Maven-based builds for Google Web Toolkit.

For owners of this book, more information, code samples, and a free ebook are available from www.manning.com/GWTinPractice

"A true hands-on manual for GWT"

—Edmon Begoli
Oak Ridge National Laboratory

"Cooper and Collins live and breathe GWT—their code is spotless."

—Andrew Grothe
Triware Technologies Inc.

"Expertly explains the genius of this technology—a real gift."

—Peter Pavlovich, Kronos Inc.

"The more complex your project, the more critical is this information."

—Ara Abrahamian, NSI ltd

"The perfect guide to GWT in the real world."

—Devon Hillard
DigitalSanctuary.com

ISBN-13: 978-1933988290
ISBN-10: 1933988290
54499
9 781933 988290

MANNING

$44.99 / Can $44.99 [INCLUDING eBOOK]
$27.50 PDF ebook at Manning.com