3.6. Handling events

You now have the widgets on the screen in `BasicProject`, the layout is (nicely) structured via the various panels you're using, and you're displaying it to the user. What happens next is the user will start clicking things, typing, and generally trying to interact; a call to the server might return with data, or the browser might even be closing. Dealing with all these interactions is the process of handling events.

Definition

An *event* is an indication that something has happened. If your application is interested in a particular event, then you need to attach a relevant handler to the widget that fires the event.

We'll look at the following in this short section:

- What events are
- How to handle events
- How to prevent the browser from handling events itself—useful if you want to stop text selection or image dragging

In chapter 14 we'll go into events in a lot more detail, including how to preview, cancel, and prevent them, as well as look at how to create your own events (which you might want to do for various reasons, including if you're implementing an `EventBus-` style architecture). For now, we'll define what events are and how to handle them.

3.6.1. What are events?

When we think of an event in GWT, we're thinking about an indication that something has happened. Conceptually, GWT contains two types of events: browser (also known as native) and logical events. A native event is one that's raised by the browser. It might come from the DOM indicating that someone has clicked a button or that an image has loaded. Or it might come from the browser when someone tries to close the browser window or resize it, or when an Ajax call to the server returns, or something else that the browser handles.

Logical events are those that mean something specific to a widget or application. For example, when you change tabs in a `TabPanel`, a `SelectionEvent` is raised by GWT.

Both types of event are treated the same in GWT, and that's through event handlers.

3.6.2. Handling events

Both native and logical events are handled in GWT by an event handler added to the object that could receive the event.

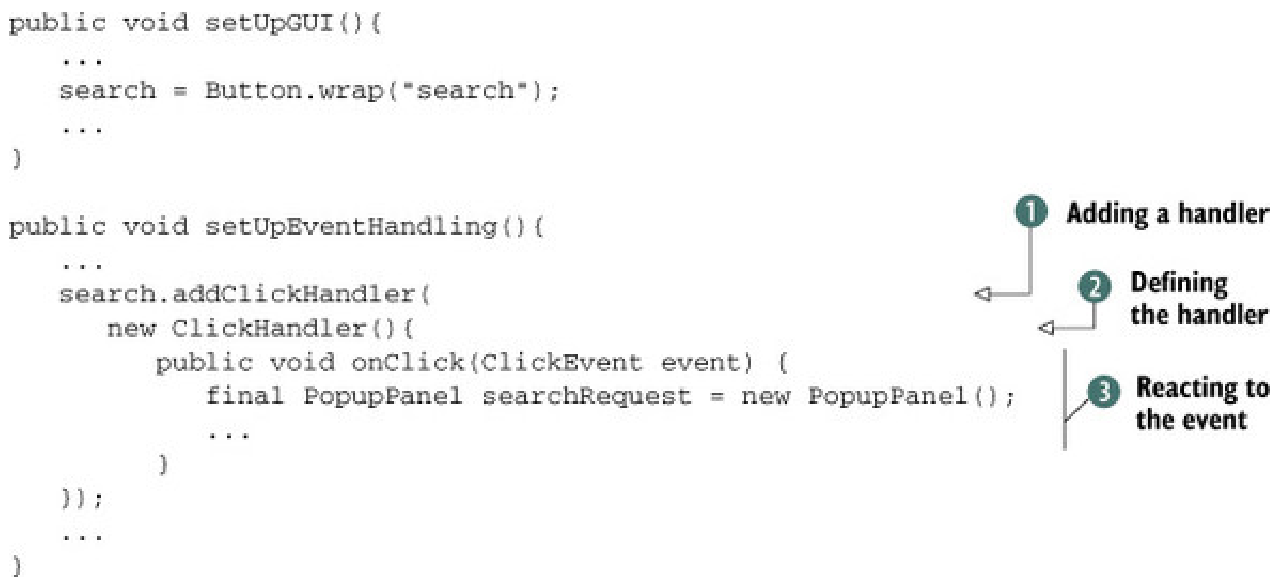
Event listener vs. event handler

GWT 2 uses an event handler model, for example, `ClickHandler`. You may see legacy code using an old event listener model, for example, `ClickListener`; that model is deprecated, and you should move the code to the handler model.

The handler model gives greater flexibility to access details about the event.

The next listing shows how to handle a click event on the example application's Search button by attaching a click handler to the button.

Listing 3.5. Adding a `ClickHandler` to a Button



```
public void setUpGUI(){
    ...
    search = Button.wrap("search");
    ...
}

public void setUpEventHandling(){
    ...
    search.addClickHandler(
        new ClickHandler(){
            public void onClick(ClickEvent event) {
                final PopupPanel searchRequest = new PopupPanel();
                ...
            }
    });
    ...
}
```

❶ Adding a handler
❷ Defining the handler
❸ Reacting to the event

Handling events follows the same, simple pattern in GWT. You add a specific event handler to the object that will receive the event. That handler will include an `onEvent` method specific to the event— `onClick` in the case of `ClickHandler` —which takes as its parameter the Java representation of the event.

This event object often has various methods that allow you to retrieve information about the event. For example, the `MouseWheelEvent` object has `isNorth/isSouth` methods allowing you to know which way the mouse wheel was spun and a `getDeltaY` method to know how far it was spun. Logical events have similar useful methods; the `SelectionEvent` has a `getSelectedItem` method so you can tell what item was selected.

In listing 3.5 you attach a `ClickHandler` ❷ through the `addClickHandler` method ❶ on a `Button` . The required `onClick` method ❸ handles a `ClickEvent` , though that event object doesn't have any other methods, so you only know a click has happened (if you're interested in which mouse button was clicked, you should use `MouseDown-Handler` instead).

Native events are all raised by the browser and trapped by GWT for you—click a button, and GWT intercepts the DOM click event, creates the GWT `ClickEvent` , and fires it at all the click handlers attached to that button.

In listing 3.5 you added a click handler to the button as an anonymous class—a fairly common approach in Java, meaning you defined it locally as an expression. You can quite easily take a named approach if you want, such as shown here:

```
class ButtonClickHandler implements ClickHandler{

   public onClick(ClickEvent event){...]

}

clicker = new ButtonClickHandler();

Button search = new Button("search");

search.addClickHandler(clicker);
```

To remove an event handler, regardless of whether it's an anonymous or named class, you need access to its `HandlerRegistration` . And you only get that when you add the handler to the widget, so if you know you'll need it later, you need to slightly amend your code to save it, as follows:

```
HandlerRegistration clickHandlerRegistration = search.addClickHandler( ... );
```

Having saved the reference to the `HandlerRegistration`, you can use it later to remove the handler:

```
clickHandlerRegistration.removeHandler();
```

The restrictions on what handlers can be added to a particular widget are usually driven by the underlying DOM element for native events or common sense for logical events. The widget's Javadoc, or your IDE's code-completion functionality, will let you see what events a widget or panel can handle.

Need to handle mouse or key events on a widget that doesn't support that?

Wrap the widget in a `FocusPanel` and add the event handlers to that panel. The event will bubble up to the `FocusPanel` to be handled there (the way GWT manages events hides the differences for you from how IE traditionally bubbles events differently to other browsers).

To the user it looks like the event is being handled by the widget.

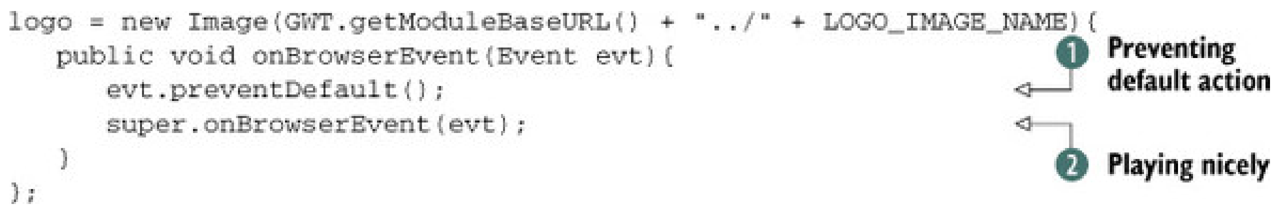
The `BasicProject` example uses this approach on the feedback tab—it places it in a `FocusPanel` in order to manage mouse-over and mouse-out events.

With the ability to handle events, the application becomes more useful. But you may also have times when too many events are being handled. Have you ever held the mouse button down and dragged the mouse over your browser screen? All the DOM elements underneath the mouse were highlighted—which would ruin any drag-and-drop functionality you might have been building. You need a way to prevent the default browser action on events.

3.6.3. Preventing the browser from handling events for you

Have you ever dragged an image from a web page to the browser location bar? The browser loaded the image as the only item on the page. This is a default browser action that you can prevent, along with others. To do this, you override a widget's `onBrowserEvent` method, called by GWT as part of its event-handling process, and tell it to prevent any default event action. Listing 3.6 shows this for the logo `Image` of this chapter's example.

Listing 3.6. Preventing default browser event handling behavior



```
logo = new Image(GWT.getModuleBaseURL() + "../" + LOGO_IMAGE_NAME){
   public void onBrowserEvent(Event evt){
      evt.preventDefault();
      super.onBrowserEvent(evt);
   }
};
```

❶ Preventing default action
❷ Playing nicely

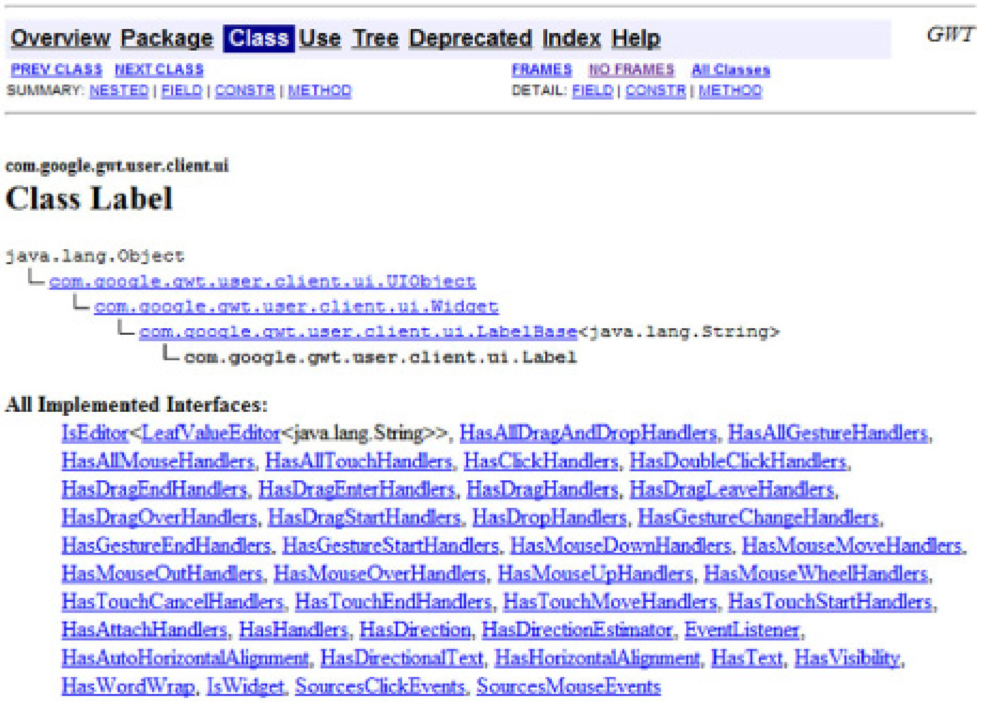
When you create the `Image` object for the `BasicProject`'s logo, you override its `onBrowserEvent` method (which every widget inherits from the base `Widget` class). To prevent the browser from running its standard functionality, you call the `prevent-Default` method on the `evt` parameter ❶. To play nicely with the GWT event-handling system, you must call the `super.onBrowserEvent` method ❷; otherwise, you wouldn't be able to handle the event later in your own code if you wanted to.

Try running the `BasicProject` application and dragging the logo to the browser location bar—nothing will happen. If you comment out the `preventDefault` call in the BasicProject.java code (in the `insertLogo` method), you can then try dragging the logo to the location bar. The behavior will be different.

Now you're at another milestone in your development; you might want to take some time to play with events in the example code. Add a few more widgets, and put some relevant event handlers on them. Often it's easiest to check out the Javadoc[4] to find out what events a widget will handle. Figure 3.6 shows part of the Javadoc entry for `Label`.

[4] For example, the Javadoc for the Label widget is http://mng.bz/0WzH.

Figure 3.6. Javadoc of `Label` showing all of the event-handling interfaces that it implements—all the events you can handle for a `Label`


Overview Package Class Use Tree Deprecated Index Help
GWT
PREV CLASS NEXT CLASS
FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD

com.google.gwt.user.client.ui

Class Label

```
java.lang.Object
  └ com.google.gwt.user.client.ui.UIObject
      └ com.google.gwt.user.client.ui.Widget
          └ com.google.gwt.user.client.ui.LabelBase<java.lang.String>
              └ com.google.gwt.user.client.ui.Label
```

All Implemented Interfaces:

IsEditor<LeafValueEditor<java.lang.String>>, HasAllDragAndDropHandlers, HasAllGestureHandlers, HasAllMouseHandlers, HasAllTouchHandlers, HasClickHandlers, HasDoubleClickHandlers, HasDragEndHandlers, HasDragEnterHandlers, HasDragHandlers, HasDragLeaveHandlers, HasDragOverHandlers, HasDragStartHandlers, HasDropHandlers, HasGestureChangeHandlers, HasGestureEndHandlers, HasGestureStartHandlers, HasMouseDownHandlers, HasMouseMoveHandlers, HasMouseOutHandlers, HasMouseOverHandlers, HasMouseUpHandlers, HasMouseWheelHandlers, HasTouchCancelHandlers, HasTouchEndHandlers, HasTouchMoveHandlers, HasTouchStartHandlers, HasAttachHandlers, HasHandlers, HasDirection, HasDirectionEstimator, EventListener, HasAutoHorizontalAlignment, HasDirectionalText, HasHorizontalAlignment, HasText, HasVisibility, HasWordWrap, IsWidget, SourcesClickEvents, SourcesMouseEvents

Looking at figure 3.6 you can generally quickly translate any of the `HasXXXXXHandlers` interfaces into writing:

```
Label testLabel = new Label("Some text");

testLabel.addXXXXXHandler(new XXXXXHandler(){

   public onXXXXX(XXXXXEvent evt){

      Window.alert("XXXXX happened");

   }

});
```

As we mentioned, we go into events in a lot more detail in chapter 14, so when you've finished exploring events yourself, we'll jump to our next topic, which is covering a specific logical event that can cause Ajax applications problems—history management.