
3.4. Creating your user interface

When you display some text to the user, provide a button for them to click, or animate a dialog box giving some information, you're using a widget. When you want to display widgets in some form of structure, then you're most likely going to be using a panel.

Definitions

A *widget* is a component of your web application with which the user may interact. A *panel* is a widget that can hold one or more other widgets in a specific visual/functional structure.

`HelloWorld`'s entry point gave perhaps the simplest example. It created a `Label` widget that's able to hold some text and added it to a panel called `RootPanel` (that represents the web page).

Widgets and panels have some subtle differences between them, so we'll look at them in separate sections—the first on widgets, the second on panels, and a third on a relatively new type of panel called a *layout panel*.

In this chapter you'll programmatically build the interface; you'll create widgets and panels in code and use Java methods on panels to add the widgets. In any real application you'll probably use a declarative approach; UiBinder, which is introduced in chapter 5, allows for separation of concerns between functionality and presentation. But to fully understand the magic behind UiBinder we feel it's necessary to look at the programmatic way first—what you learn won't be lost if you use UiBinder—and our next section starts our journey, looking at widgets.

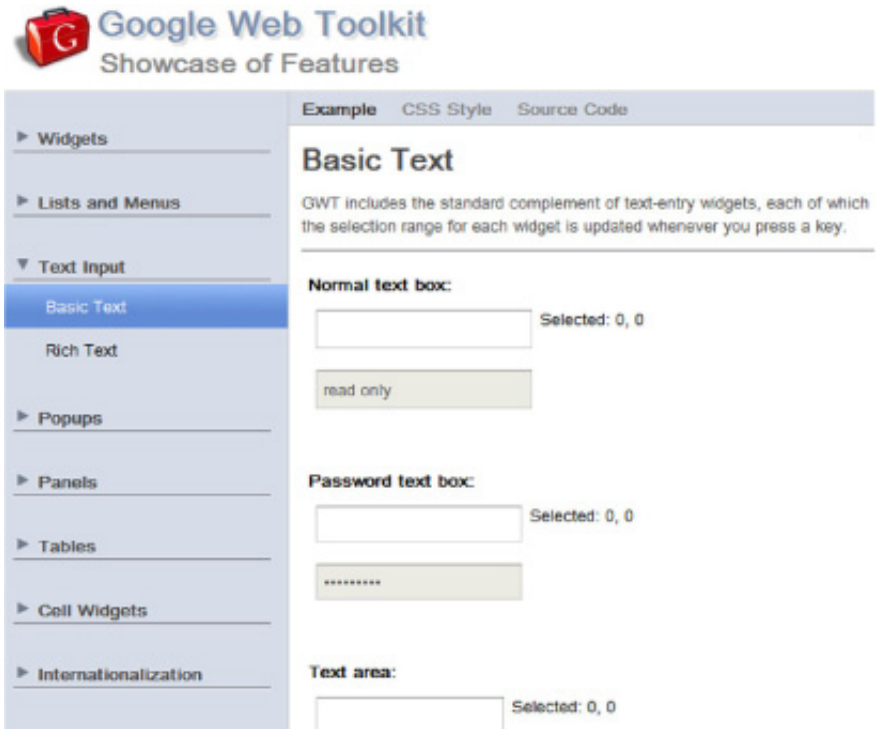
3.4.1. Presenting widgets

As you saw at the start of this section, widgets are the components that are used to interact with the user: buttons, labels, and so on. GWT has widgets that represent standard HTML/DOM objects such as images and buttons, and widgets that represent more abstract concepts, such as trees or menu bars. There's even a set of data-presentation widgets, discussed in chapter 10, that give an efficient view of large data sets. Luckily we don't need to differentiate between the types; they're all widgets and they're created and used in exactly the same way.

We'd particularly recommend running the GWT Showcase of Features that comes with the GWT download (or can be accessed online[2]) to get a good view of the types of widgets that GWT provides—see figure 3.2 for an example. You can also find third-party libraries that provide even more widgets or fancier versions of the standard ones; let's not worry about those for now because we want to concentrate on widgets in general.

[2] The GWT Showcase can be found online at http://gwt.google.com/samples/Showcase/Showcase.html.

Figure 3.2. A screenshot of GWT's Showcase of Features



In the rest of this section we'll look at the two ways that widgets can be created, as well as how they fit into a particular object hierarchy.

Creating Widgets

The example application's `setUpGui` method in the `EntryPoint` class creates GWT widgets in a couple of ways. It wraps existing widgets, as you'll see in a moment, and it creates widgets from scratch; for example, the logo image is created and inserted into the browser in the `insertLogo` method in a manner similar to this:

```
String LOGO_IMAGE_NAME= "gwtia.png";

Image logo = new Image(GWT.getModuleBaseURL()+"../" + LOGO_IMAGE_NAME);
```

In the Java code, you create a Java object of the type `Image` and pass the location of the image resource into the constructor. The `GWT.getModuleBaseURL` method returns the path to your web application, onto which the image filename, stored as a constant for future maintainability, is appended to give the full path, showing the logo as in figure 3.3.

Figure 3.3. The application's logo created as a GWT `Image` widget

You may be wondering why you go up a directory first to find the image file. The GWT-generated code goes into the BasicProject folder in the war; the image, though, is directly under the war, one folder up from our code. You keep the source image at that level to ensure it's not overwritten by GWT during compilation. You can place it anywhere in the war file that suits you as long as you link correctly.

Behind the scenes, widgets have a dual existence. In your Java code they're pure Java objects, and in the browser they're DOM elements. The Java `Image` object previously shown becomes the following DOM object when you're running your application (where `xxxxxxx` is the path to the JavaScript code of the application):

```
<image src="xxxxxxxxx/../gwtia.png">
```

Do you need to know this? Most of the time you don't, because GWT handles all this for you. But remembering this fact will help you understand why you sometimes can't do things you want to do, or think you should be able to do—the DOM restricts you.

A good example of DOM restrictions is the standard `FileUpload` widget. It's common to hear people ask why you can't set the default text

of this widget to a specific filename. You can't do this because the GWT `FileUpload` widget is implemented by the DOM `FileUpload` element—and the DOM stops you from programmatically selecting a filename for good security reasons.

Note

Always remember that your Java code gets compiled to JavaScript that manipulates the browser DOM. Although GWT can support you in pushing the boundaries of Ajax applications, it can't break the rules of the DOM.

You can also wrap HTML elements that already exist on the web page. This method is particularly useful if you want to incrementally add GWT to your page. You might already have a button that when clicked starts some functionality, and you're moving that functionality into GWT. You can wrap that existing button in your application, and when it's clicked, your new GWT functionality starts.

To wrap an existing element, you use the closest widget's static `wrap` method. Not all widgets support this approach, so you need to check the API. When a widget allows this, it also checks to see that the element you're trying to wrap is of the DOM element it normally represents.

The `Button` widget allows you to wrap an existing `BUTTON` element. In `Basic-Project` the existing Search button in the application's HTML page is wrapped in the `wrapExistingSearchButton` method as follows:

```
Button search = Button.wrap(DOM.getElementById("search"));
```

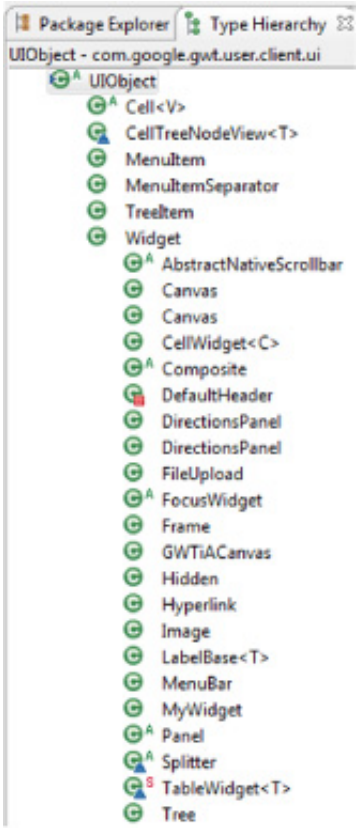
You need to pass the `wrap` method the relevant DOM element, and you'll find that through the `DOM.getElementById` method, passing in the `id` of the element you're trying to find. You might remember in listing 3.2 that you defined the button in the HTML page to have an `id` of `search` —this is the value you use here.

It's worth noting that wrap methods don't exist for all widgets, and they don't wrap any existing event handling on the DOM element—so you'll need to add event handling yourself in your GWT application. Also, if the wrapped element is removed from the DOM, you need to watch for that and call the `RootPanel.detachNow` method to ensure no memory leaks are introduced (this is covered more in chapter 4).

Widget Hierarchy

A benefit of the Java/DOM dual view of widgets is that on the Java side they live within a hierarchy, giving predictable behavior through inheritance. You can find the standard GWT widgets in the `com.google.gwt.user.client.ui` package, and the first few levels of the hierarchy are shown in figure 3.4.

Figure 3.4. First three layers of the GWT widget hierarchy starting with `UIObject` descending to, for example, `Tree`



Every widget is a subclass of `UIObject`, which provides methods for setting a widget's size (for example, `setWidth`, `setHeight`, `SetSize`, and `SetPixelSize`), getting its size and position (for example, `getWidth` and `getAbsoluteLeft`), and setting its visibility (`setVisible`).

`Widget` is a subclass of `UIObject` and extends the functionality by adding the ability to deal with attaching and detaching widgets to/from the browser DOM and hooking into the GWT event system. By managing these two aspects, GWT removes the chance of memory leaks that can often sneak into Ajax applications when manipulating the DOM directly. If you're interested in the mechanics of attaching/detaching widgets, we cover the topic in more detail in chapter 4.

Next in the widget hierarchy you'll find some real widgets, such as `FileUpload`, `Image`, and `Tree`. You may be wondering where widgets for button, text box, radio button, and so on are. These are all hiding under the `FocusWidget` class, along with some GWT additional widgets such as `PushButton`, `ToggleButton`, `RichTextArea`, and others.

Two subclasses of `Widget` deserve a little more explanation: `Composite` and `Panel`. `Composite` widgets are widgets typically made from two or more other widgets that you wish to treat as a single widget. You'll probably spend a large part of your development time creating these types of widgets because they'll form the valuable and reusable components of your application. We'll look at how to create `Composite` widgets in chapter 4, but GWT provides some as standard. These include, among others, `Caption-Panel` (a panel that has a caption), `DatePicker` (for picking dates), and `TabPanel`.

`Panel`s, from a Java perspective, are a subset of `Widget`, but they have some special properties that earn them a section all their own.

3.4.2. Organizing layout with Panels

A `Panel` is a special case of `Widget`: it can hold one or more other widgets (which themselves might be panels holding other widgets, or panels that hold other widgets or panels that hold other widgets or panels, that hold … well, you get the gist). So panels give you a way of organizing the presentational structure of other panels and widgets. To do that, a `Panel` introduces the `add` and `remove` methods to the

basic `Widget` class.

The way the `add` method is implemented gives a panel its specific functionality. For example, the `add` method in a `SimplePanel` allows only one widget to be added, throwing an exception if you attempt to add a second widget, whereas a `Vertical-Panel` allows one or more widgets to be added, appearing in a visual vertical list that grows downward with each new added widget; by contrast, any number of widgets can be added to a `DeckPanel` but only one is visible at a particular time.

Panels are also constrained by their underlying DOM implementation—panels have the same dual Java/DOM existence as normal widgets—remember they're a special type of widget. Typically `div` - and `Table` -based panels, such as `SimplePanel` and `VerticalPanel` , are used for structuring internals of more complicated widgets, for example, the composite widgets we'll look at in chapter 4. For application structure, it's preferable to use layout panels, such as `DockLayoutPanel` and `TabLayoutPanel` , which are based on box model dimension constraints. Doing that, you let the browser do the hard work of rendering and layout, making these panels super-fast—but this means you have to put the browser into its standards-compliant mode (more about this and layout widgets in general in chapter 4).

Using your browser's standards mode?

If you're using the standards mode of browsers, you should consider using layout panels for a number of different panels to avoid some layout gremlins.

For `DockPanel` use `DockLayoutPanel` , for `SplitPanel` use `SplitLayoutPanel` , and for `StackPanel` use `StackLayoutPanel` .

For `VerticalPanel` use `FlowPanel` , which by default breaks each new item onto the next line.

For `HorizontalPanel` use `FlowPanel` and set the CSS `float` property to `left` on all added widgets (to suppress the behavior of `FlowPanel` previously described for `VerticalPanel`).

Standards mode means a browser will render DOM and so on, adhering to W3C and IETF standards; some browsers allow a "quirks" mode for backward compatibility for cases where they may have implemented rendering slightly differently.

As we did for widgets, let's look at how to create panels. We'll also give you a flavor of the panel types that GWT provides.

Creating a Panel

In the `BasicProject` example you use five panels, `TabLayoutPanel` , `HTMLPanel` , `FocusPanel` , `VerticalPanel` , and `RootPanel` , in the following ways:

- A `TabLayoutPanel` is used for structuring the main display of the application. We could have used a `TabPanel` instead because it's functionally equivalent, but we want to harness the flexibility and speed of layout panels for the application structure.
- The example's feedback object on the right of the screen (see figure 3.2) will be a `VerticalPanel` holding two label widgets.
- The `VerticalPanel` will be wrapped in a `FocusPanel` , which listens to all mouse events because `VerticalPanel` doesn't allow that directly. The application will listen for *mouse over* and *mouse out* events—you grow the panel when the mouse is over it and shrink it when the mouse moves out of it.
- You extract the HTML content of `BasicProject` 's HTML page's `div` s with `id` s `home, product` , and `contact` into `HTMLPanel` s (panels that hold HTML content).
- You place all three `HTMLPanel` s into a `TabLayoutPanel` (a panel that shows a number of tabs and one of its enclosed panels depending on which tab button is selected).

- You use the `RootPanel` to place all these panels on the browser page.

The tabs are built up as shown in listing 3.3 (from the `BasicProject` 's `buildTab-Content` method; `getContent` is another helper method, and we'll look at that when we consider manipulating the page a little later in this chapter).

Listing 3.3. Creating part of the panel structure for the chapter example

```
static final int DECK_HOME = 0;
HTMLPanel homePanel;
HTMLPanel productsPanel;
HTMLPanel contactPanel;
TabLayoutPanel content

private void buildTabContent(){
   homePanel = new HTMLPanel(getContent("home"));
   productsPanel = new HTMLPanel(getContent("product"));
   contactPanel = new HTMLPanel(getContent("contact"));

   content = new TabLayoutPanel(20, Unit.PX);

   content.add(homePanel, "home");
   content.add(productsPanel, "products");
   content.add(contactPanel, "contacts");

   content.selectTab(DECK_HOME);
}
```

1 Creating HTMLPanels

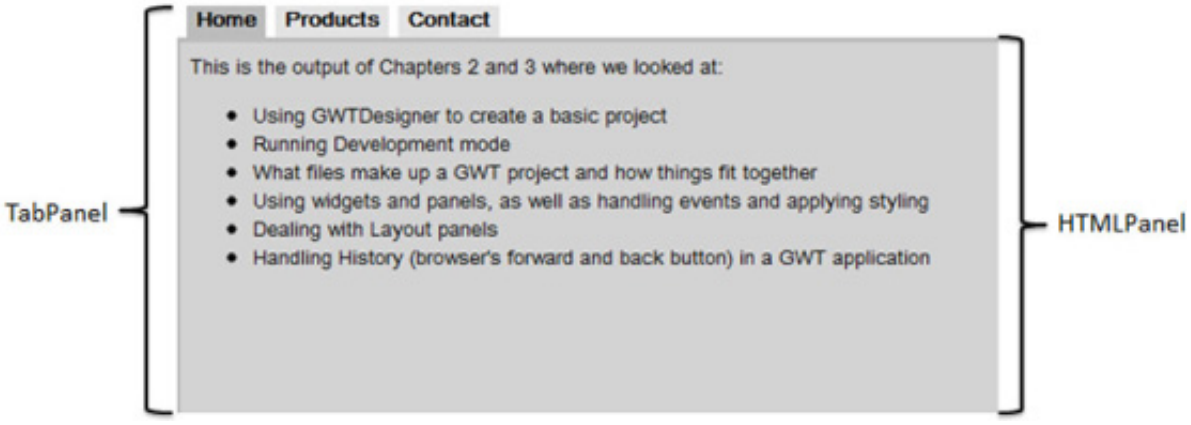
2 Creating TabPanel

3 Adding content to TabPanel

4 Which TabPanel content to show first

Figure 3.5 shows the tab panel created from the code in listing 3.3, and the code creates new HTML panels from content already in the HTML (1), creates a new `TabLayoutPanel` (2), and then adds the content created in (1) to that new panel (3). Finally, it selects the Home tab (4). (If you look at the downloaded code you'll see that the hardcoded strings in listing 3.3, for example, "home," are replaced with a more flexible approach, but we used the hardcoded strings for brevity.)

Figure 3.5. A GWT `TabPanel` whose content is made up of several `HTMLPanel` s—one per tab



When you add a widget to a panel, different things happen depending on whether you're thinking about the Java or DOM world. In the Java world, the panel contains a variable to represent the widget(s) it holds, and the widget is added to that variable if the rules of the panel are followed (it's in Java that `SimplePanel` is restricted to having only one widget). Over in the DOM world, the DOM is manipulated so that the widget is inserted into the appropriate DOM structure. Table 3.2 shows the two views when adding a `Button` to a `SimplePanel` and then adding that panel to the `RootPanel` .

Table 3.2. Comparing Java code to DOM when building up a simple panel with a button added to it and adding the panel to the HTML page

Java code	DOM representation

```
SimplePanel holder = new
SimplePanel();

Button stop = new Button("Stop");

holder.add(stop)

RootPanel.get().add(holder)
```

```
<div></div>

<button type="button">
   Stop
</button>

<div>
   <button type="button">
      Stop
   </button>
</div>

<html>
   <head>
      :
   </head>
   <body>
      :
      <div>
         <button type="button">
            Stop
         </button>
      </div>
      :
   </body>
</html>
```

The `SimplePanel` is represented by a `div` element in the DOM and a `Button` as a `button` DOM element. When we add the button to the panel, we're setting a private field in the Java representation of the `SimplePanel` to be the Java representation of the `Button`. Over in the DOM world we're physically inserting the DOM representation of the button into the DOM representation of the simple panel.

In the final row of table 3.2 we're inserting the `SimplePanel` onto the browser page. In Java it's a simple call to the `get` method of the `RootPanel` and adding the `SimplePanel` to the result. Over on the right-hand side you can see that this Java code translates to some more DOM manipulation, inserting the DOM representation of the `SimplePanel` into the `body` of the HTML page.

So, that's how panels work and differ slightly from widgets. In chapter 4 we'll peek under the hood a little and see how GWT manages widgets and panels in more detail, but for now we'll continue by considering what types of panels GWT provides.

Types of Panels

GWT provides many types of panels, and, as with widgets, GWT's Showcase is a good place to get familiar with them—anything we write in this book would soon be out of date with new panels being added all the time. Instead of discussing the Java hierarchy of panels, as we did with widgets, it's better to think of a panel as sitting in one of five buckets:

- Simple panels that can take only one widget
- Split panels that contain two widgets on either side of a splitter bar that can be dragged by the mouse
- Table-based panels— `Grid` and `FlexTable`; other table-based panels are known as cell panels under the complex panel category (these cell panels shouldn't be confused with the `cell` widgets discussed in chapter 10, easy as it is to do)

- So-called complex panels
- Layout panels

Within the class of simple panels you'll find `FocusPanel` (which can handle all browser focus events), `FormPanel` (for standard HTML forms), `PopupPanel`, and `ScrollPanel`. All work as you'd expect, although you should take care with `ScrollPanel` to explicitly set its size. (If you don't, it will expand to the size of the widget you add, and no scroll bars will appear.)

The simple panel class also includes `LazyPanel` and `DecoratorPanel`. `LazyPanel` defers some of its computation until it's required, which might be useful if you're looking to speed up your application. `DecoratorPanel` is GWT's way of adding nice rounded corners to widgets.

Split panels allow you to add two widgets, one on either side of a splitter bar that you can drag—a horizontal one and a vertical one come for free.

You'd use the table-based `Grid` panel if you know the dimensions of the panel aren't going to change once you've created it. On the other hand, you can use a `FlexTable` when the dimensions might change after creation (with `FlexTable` you can add as many new rows and columns as you want).

Complex panels cover `AbsolutePanel` (where the x/y position of widgets within the panel can be explicitly set), `DeckPanel` (which acts as a deck of cards showing only one widget at a time), `FlowPanel` (where widgets should flow in the direction of the locale,[3] though you need to remember what you're putting in here—adding a `Label` widget won't flow as you expect, because `Label` is implemented by a `div` element, which will force a new line unless you get funky with some CSS and set the `float` property to left—try using GWT 2.0's new `InlineLabel` instead), `HTMLPanel` (holds HTML content), `StackPanel`, and a set of cell-based (table) panels such as `VerticalPanel`, `HorizontalPanel`, and `DockPanel`.

[3] For example, left to right for English or right to left for Arabic.

Some panels have animation built into them. Panels and widgets that implement the `HasAnimation` interface, such as `Tree`, `SuggestBox`, `PopupPanel`, `MenuBar`, `DisclosurePanel`, and `DeckPanel`, can be set to animate as changes are made to them, for example, opening a tree branch. If you're running this chapter's main example, you'll see the pop-up, which appears when you click the Search button, animate into view. It does this because you call `setAnimationEnabled(true)` when you create it, and the GWT designers had built this animation into the panel.

The final bucket of panels we mention are `Layout` panels. These provide some powerful layout functionality and are driven by style constraints—all you need to know about that for now is that it makes them blisteringly fast. In older versions of GWT we had `DockPanel`, `TabPanel`, `HorizontalSplitPanel`, and `StackPanel`. The latest versions of GWT still have those but also provide `Layout` panel equivalents, such as `DockLayoutPanel`, `TabLayoutPanel`, `SplitLayoutPanel`, and `StackLayoutPanel`. As we've mentioned, `Layout` panels are best suited to the outer structure of your application, and we'll look at them in some more detail in section 4.6, whereas the older-style non-layout versions are more often used for structuring a widget's internals (but see the sidebar "Using your browser's standards mode?" to see some suggestions on using non-layout panels when using the browser's standards mode).

Once you've created widgets and panels, you need to add them to the browser DOM; otherwise, your user will never be able to see or use them. That's done, as you'll see next, by manipulating the page.