Chapter 12

Using Selection Widgets

In Chapter 11, you saw how fields could have constraints placed on them to limit possible input, such as numeric-only or phone-number-only. These sorts of constraints help users “get it right” when entering information, particularly on mobile devices with cramped keyboards.

Of course, the ultimate in constrained input is to allow selection only from a set of items, such as a group of radio buttons. Classic UI toolkits have list boxes, combo boxes, drop-down lists, and the like for that very purpose. Android provides many of the same sorts of widgets, plus others of particular interest for mobile devices (e.g., the Gallery for examining saved photos).

Moreover, Android offers a flexible framework for determining which choices are available in these widgets. Specifically, Android offers a framework of data adapters that provides a common interface for selection lists, ranging from static arrays to database contents. Selection views—widgets for presenting lists of choices—are handed an adapter to supply the actual choices.

Adapting to the Circumstances

In the abstract, adapters provide a common interface to multiple disparate APIs. More specifically, in Android’s case, adapters provide a common interface to the data model behind a selection-style widget, such as a list box. This use of Java interfaces is fairly common (e.g., Java/Swing’s model adapters for JTable), and Java is far from the only environment offering this sort of abstraction (e.g., Flex’s XML data-binding framework accepts XML inlined as static data or retrieved from the Internet).

Android’s adapters are responsible not only for providing the roster of data for a selection widget, but also for converting individual elements of data into specific views to be displayed inside the selection widget. The latter facet of the adapter system may sound a little odd, but in reality, it is not that different from other GUI toolkits’ ways of overriding default display behavior. For example, in Java/Swing, if you want a JList-backed list box to actually be a checklist (where individual rows are a check box plus label, and clicks adjust the state of the check box), you inevitably wind up calling
setCellRenderer() to supply your own ListCellRenderer, which in turn converts strings for the list into JCheckBox-plus-JLabel composite widgets.

The easiest adapter to use is ArrayAdapter. You simply wrap one of these around a Java array or java.util.List instance, and you have a fully functioning adapter:

```java
String[] items={"this", "is", "a", "really", "silly", "list"};
new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, items);
```

One flavor of the ArrayAdapter constructor takes three parameters:

- The Context to use (typically this will be your activity instance)
- The resource ID of a view to use (such as a built-in system resource ID, as shown in the preceding example)
- The actual array or list of items to show

By default, the ArrayAdapter will invoke toString() on the objects in the list and wrap each of those strings in the view designated by the supplied resource. android.R.layout.simple_list_item_1 simply turns those strings into TextView objects. Those TextView widgets, in turn, will be shown in the list, spinner, or whatever widget uses this ArrayAdapter. If you want to see what android.R.layout.simple_list_item_1 looks like, you can find a copy of it in your SDK installation—just search for simple_list_item_1.xml.

In Chapter 13, you’ll see how to subclass an adapter and override row creation, to give you greater control over how rows appear.

## Lists of Naughty and Nice

The classic list box widget in Android is known as ListView. Include one of these in your layout, invoke setAdapter() to supply your data and child views, and attach a listener via setOnItemSelectedListener() to find out when the selection has changed. With that, you have a fully functioning list box.

However, if your activity is dominated by a single list, you might consider creating your activity as a subclass of ListActivity, rather than the regular Activity base class. If your main view is just the list, you do not even need to supply a layout—ListActivity will construct a full-screen list for you. If you do want to customize the layout, you can, as long as you identify your ListView as @android:id/list, so ListActivity knows which widget is the main list for the activity.

For example, here is a layout pulled from the Selection/List sample project, a simple list with a label on top to show the current selection: