Implementing Android User Interfaces

Contents

Chapter 1

Views and Layouts

Objectives After this section, you will be able to:

- Describe the difference between the `View` and `ViewGroup` classes.
- List four primary `ViewGroup` subclasses and describe when you would use them.
- List the steps required to define an activity's layout either programmatically or declaratively.

1.1 The `View` Class Hierarchy

`View` is the base class for all Android visual components (sometimes known as *widgets*).

- It defines properties and methods applicable to all types of visual components.
- Many subclasses of `View` exist for standard visual components, such as `TextView`, `EditText`, `Button`, `ImageView`, `ImageButton`, etc.

In addition to the standard `View` subclasses, you can create your own custom components by either:

- Extending existing `View` subclasses — for example, creating a specialized subclass of `EditText`
- Creating custom subclasses of `View`
- Combining several standard or custom views into a custom composite component

---

The "Custom Components" section of the *Developer's Guide* provides more details about creating custom view components.

1.2 ViewGroups: Combining Views into a Layout

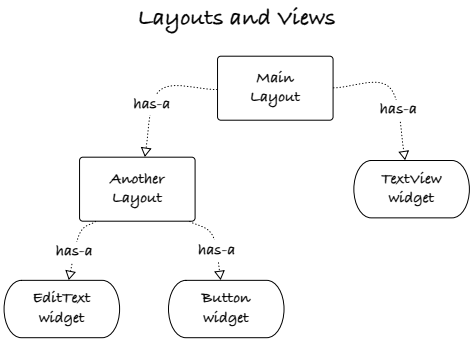

Figure 1.1: Layouts and Views

In Android, `ViewGroup` is the base class for all *layout managers*, which controls the size and position of views in a layout.

- `ViewGroup` is a subclass of `View`.
- You can add one or more views as *children* to a `ViewGroup`.
- You can nest `ViewGroups` within other `ViewGroups` to create complex layouts.

Child views use *layout parameters* to request how the layout manager should display them.

- Each `ViewGroup` subclass supports its own set of *layout parameters*.
- Some parameters, such as `layout_height` and `layout_width`, are common to all managers.

`Views` and `ViewGroups` are an example of the *Composite design pattern*.

1.3 Common Layout Managers

Android provides several different `ViewGroup` subclasses. Some of the more commonly used ones are:

`LinearLayout`
: A `LinearLayout` aligns all its children in a single direction — either vertically or horizontally, depending on how you define the layout's `orientation` attribute.

`RelativeLayout`
: A `RelativeLayout` lets child views specify their position relative to the parent view or to each other. For example, you can align two elements by right border, or make one below another, centered in the screen, centered left, and so on.

`TableLayout`
: A `TableLayout` positions its children into rows and columns. `TableRow` objects are the child views of a `TableLayout`, with each `TableRow` defining a single row in the table. Each row has zero or more cells, each of which is defined by any kind of other `View`.

`FrameLayout`
All child elements of a `FrameLayout` are pinned to the top left corner of the screen; you cannot specify a different location for a child view. Subsequent child views will simply be drawn over previous ones, partially or totally obscuring them (unless the newer object is transparent). A `FrameLayout` is often used to contain a set of views whose visibility is controlled programmatically at runtime so that only one is displayed at a time.

1.4 Programmatic vs. Declarative Layouts

You can provide an activity's layout *programmatically* or *declaratively*.

Programmatically:

- Instantiate `View` and `ViewGroup` objects directly in the Java code.
- Set their properties and layout parameters programmatically.
- Invoke `Activity.setContentView(View)` with a reference to your top-level `ViewGroup` to display the layout.

Declaratively:

- Create an XML *layout resource*.
- Use XML elements to indicate the `View` and `ViewGroup` objects to include.
- Use XML attributes to set their properties and layout parameters.
- Invoke `Activity.setContentView(int)`, specifying your layout resource.
- Your activity *inflates* the layout, instantiating the layout XML file into its corresponding `View` objects.

You can use either or both techniques to define your layouts. In general, the declarative approach is preferred for the following reasons:

- The declarative approach requires less code.
- The Eclipse ADT plugin provide a graphical layout editor allowing you to preview your layout as you create it.
- It's easier to provide alternate layouts to support multiple screen sizes and orientations.

1.5 Topic Summary

You should now be able to:

- Describe the difference between the `View` and `ViewGroup` classes.
- List four primary `ViewGroup` subclasses and describe when you would use them.
- List the steps required to define an activity's layout either programmatically or declaratively.

Chapter 2

Handling User Interface Events

Objectives After this section, you will be able to:

- List some of the common user interface events and how to detect their occurrence.
- Implement an anonymous local class as an event handler.
- Implement an event handler using an existing class.
- Select which strategy for implementing an event handler is appropriate for your code.

2.1 Event Handling in Java

It's common to want to detect an *event* that occurs in an object and to provide some code that *handles* the event.

- For example, detecting when the user clicks a button and executing some code in response.

Java typically follows the *Observer design pattern* to provide a framework for handling events.

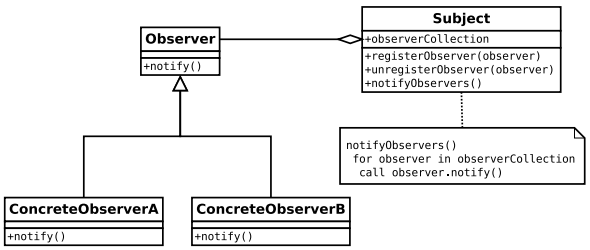

Figure 2.1: Observer Pattern

- A class of object, such as *Button*, supports one or more observable events, such as a click event. It is known as the *subject*.

- To support arbitrary handlers, it defines one or more Java interfaces declaring a set of handler methods (also known as *callback* methods).
- Objects that want to detect the events, known as *observers*, implement the defined interfaces.
- The subject class provides methods for *registering* observers.
- When the event occurs, the subject *notifies* all registered observers by invoking the appropriate callback method defined by the interface.

2.2 Common User Interface Events and Their Handlers

In Android, the `View` base class defines several interfaces for providing handlers for different types of user events:

Interface	Callback Method	Description
`OnClickListener`	`onClick()`	The user either touches the item (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses the suitable "enter" key or presses down on the trackball.
`OnLongClickListener`	`onLongClick()`	The user either touches and holds the item (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses and holds the suitable "enter" key or presses and holds down on the trackball (for one second).
`OnFocusChangeListener`	`onFocusChange()`	The user navigates onto or away from the item, using the navigation-keys or trackball.
`OnKeyListener`	`onKey()`	The user is focused on the item and presses or releases a key on the device.
`OnTouchListener`	`onTouch()`	The user performs an action qualified as a *touch event*, including a press, a release, or any movement gesture on the screen (within the bounds of the item). Intended for fine-grain event handling such as drawing or fling detection.

Note
All of these interfaces are defined as *inner interfaces* within the `View` class.

2.3 Using an Anonymous Local Class as an Event Handler

Common practice in object oriented design is to implement a special-purpose listener class for each listener.

- This isolates the custom event handling for a specific subject, rather than overloading a class with multiple responsibilities.
- This is an example of the principle of *separation of concerns*.

In Java, such "one-use" classes are often implemented as an *anonymous local class*.

- An anonymous class is defined and instantiated in a single expression using the `new` operator and specifying the base class or implemented interface. For example:

```
Button button;
// ...
button.setOnClickListener( new View.OnClickListener() {
        public void onClick(View v) {
```

```
                // Code for handling the button click
        }
});
```

2.4 The Drawback of Using Anonymous Local Classes in Android

The instance of the listener object consumes memory.

- As an independent object the memory consumption is slightly more than if you had implemented the callback interface on an existing class, such as the activity.

Once all references to the listener object are released, the listener is eventually garbage collected.

- Garbage collection is slow and runs at unpredictable times, which can cause random slowdowns of your app.

The anonymous class definition consumes memory.

- Although you don't provide an explicit name for the anonymous local class, the Java compiler still generates a name for the class, then creates an instance of that class.
- The class definition is loaded by the virtual machine when it first needs to create an instance of the class.
- Each class definition consumes memory. Even a minimal class definition in Android consumes approximate 1KB of memory.

Tip
You need to determine the balance between good object oriented design (OOD) and minimizing memory usage for your application.

2.5 Using an Existing Class as an Event Handler

A common practice in Android is to use the activity as an event listener. For example:

```
public class MyActivity extends Activity implements OnClickListener {
        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);

                // Inflate layout
                setContentView(R.layout.main);

                // Find Button object
                Button myButton = (Button) findViewById(R.id.my_button);

                // Register activity as click listener
                myButton.setOnClickListener(this);
        }

        public void onClick(View v) {
                // Handle button click
        }
}
```

2.6 Using a Event Handler for Multiple Subjects

What if you need to handle the same type of event for multiple subjects?

- For example, what if your layout has multiple buttons?
- OOD principles would suggest a separate handler class for each subject.

As an alternative, you can register the same listener for each subject.

- Your handler receives a reference to the `View` receiving the event.
- In your handler, you can `switch` based on the ID of the `View`. For example:

```
public class MyActivity extends Activity implements OnClickListener {
        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);

                // Inflate layout
                setContentView(R.layout.main);

                // Find Button objects
                Button firstButton = (Button) findViewById(R.id.first_button);
                Button secondButton = (Button) findViewById(R.id.second_button);

                // Register activity as click listener
                firstButton.setOnClickListener(this);
                secondButton.setOnClickListener(this);
        }

        public void onClick(View v) {
                int id = v.getId();
                switch (id) {
                case R.id.first_button:
                        // Handle firstButton click
                        break;
                }
                case R.id.second_button:
                        // Handle secondButton click
                        break;
                }
                default:
                        // Unknown clicked view? We shouldn't get here.
        }
}
```

2.7 Topic Summary

You should now be able to:

- List some of the common user interface events and how to detect their occurrence.
- Implement an anonymous local class as an event handler.
- Implement an event handler using an existing class.
- Select which strategy for implementing an event handler is appropriate for your code.