Using Intents to Launch Activities

The most common use of Intents is to bind your application components. Intents are used to start, stop, and transition between the Activities within an application.

The instructions given in this section refer to starting new Activities, but the same rules generally apply to Services as well. Details on starting (and creating) Services are available in Chapter 8.

To open a different application screen (Activity) in your application, call startActivity, passing in an Intent, as shown in the snippet below.

```
startActivity(myIntent);
```

The Intent can either explicitly specify the class to open, or include an action that the target should perform. In the latter case, the run time will choose the Activity to open, using a process known as

"Intent resolution."

The startActivity method fi nds, and starts, the single Activity that best matches your Intent.

When using startActivity, your application won't receive any notifi cation when the newly launched Activity fi nishes. To track feedback from the opened form, use the startActivityForResult method described in more detail below.

Explicitly Starting New Activities

You learned in Chapter 2 that applications consist of several interrelated screens — Activities — that must be included in the application manifest. To connect them, you may want to explicitly specify which Activity to open.

To explicitly select an Activity class to start, create a new Intent specifying the current application context and the class of the Activity to launch. Pass this Intent in to startActivity, as shown in the following code snippet:

```
Intent intent = new Intent(MyActivity.this, MyOtherActivity.class);
startActivity(intent);
```

After calling startActivity, the new Activity (in this example, MyOtherActivity) will be created and become visible and active, moving to the top of the Activity stack.

Calling finish programmatically on the new Activity will close it and remove it from the stack. Alternatively, users can navigate to the previous Activity using the device's Back button.

Implicit Intents and Late Runtime Binding

Implicit Intents are a mechanism that lets anonymous application components service action requests. When constructing a new implicit Intent to use with startActivity, you nominate an action to perform and, optionally, supply the data on which to perform that action.

When you use this new *implicit* Intent to start an Activity, Android will — at run time — resolve it into the class best suited to performing the action on the type of data specifi ed. This means that you can create projects that use functionality from other applications, without knowing exactly which application you're borrowing functionality from ahead of time.

For example, if you want to let users make calls from an application, rather than implementing a new dialer you could use an implicit Intent that requests that the action ("dial a number") be performed on a phone number (represented as a URI), as shown in the code snippet below:

```
if (somethingWeird && itDontLookGood) {
Intent intent = new Intent(Intent.ACTION_DIAL,
Uri.parse("tel:555-2368"));
startActivity(intent);
}
```

Android resolves this Intent and starts an Activity that provides the dial action on a telephone number — in this case, the dialler Activity.

Various native applications provide components to handle actions performed on specifi c data. Thirdparty applications, including your own, can be registered to support new actions or to provide an alternative provider of native actions. You'll be introduced to some of the native actions later in this chapter.

Introducing Linkify

Linkify is a helper class that automagically creates hyperlinks within TextView (and TextView-derived) classes through RegEx pattern matching.

Text that matches a specifi ed RegEx pattern will be converted into a clickable hyperlink that implicitly fires start Activity(new Intent(Intent.ACTION_VIEW, uri)) using the matched text as the target URI.

You can specify any string pattern you want to turn into links; for convenience, the Linkify class provides presets for common content types (like phone numbers and e-mail/web addresses).

The Native Link Types

The static Linkify.addLinks method accepts the View to linkify, and a bitmask of one or more of the default content types supported and supplied by the Linkify class: WEB_URLS, EMAIL_ADDRESSES, PHONE_NUMBERS, and ALL.

The following code snippet shows how to linkify a TextView to display web and e-mail addresses as hyperlinks. When clicked, they will open the browser or e-mail application, respectively.

```
TextView textView = (TextView)findViewById(R.id.myTextView);
Linkify.addLinks(textView, Linkify.WEB_URLS|Linkify.EMAIL_ADDRESSES);
```

You can linkify Views from within a layout resource using the android:autoLink attribute. It supports one or more (separated by |) of the following self-describing values: none, web, email, phone, or all. The following XML snippet shows how to add hyperlinks for phone numbers and e-mail addresses:

```
<TextView
android:layout_width="fill_parent"
android:layout_height="fill_parent"
android:text="@string/linkify_me"
android:autoLink="phone|email"
/>
```

Creating Custom Link Strings

To defi ne your own linkify strings, you create a new RegEx pattern to match the text you want to display as hyperlinks.

As with the native types, you linkify the target view by calling Linkify.addLinks, but this time pass in the new RegEx pattern. You can also pass in a prefi x that will be prepended to the target URI when a link is clicked.

The following example shows a View being linkifi ed to support earthquake data provided by an Android Content Provider (that you will create in the next Chapter). Rather than include the entire schema, the linkify pattern matches any text that starts with "quake" and is followed by a number. The content schema is then prepended to the URI before the Intent is fi red.

```
int flags = Pattern.CASE_INSENSITIVE;
Pattern p = Pattern.compile("\\bquake[0-9]*\\b", flags);

Linkify.addLinks(myTextView, p,
"content://com.paad.earthquake/earthquakes/");
```

Linkify also supports TransformFilter and MatchFilter interfaces. They offer additional control over the target URI structure and the defi nition of matching strings, and are used as shown in the skeleton code below:

```
Linkify.addLinks(myTextView, pattern, prefixWith,
new MyMatchFilter(), new MyTransformFilter());
```

Using the Match Filter

Implement the acceptMatch method in your MatchFilter to add additional conditions to RegEx pattern matches. When a potential match is found, acceptMatch is triggered, with the match start and end index (along with the full text being searched) passed in as parameters.

The following code shows a MatchFilter implementation that cancels any match that is immediately preceded by an exclamation mark.

```
class MyMatchFilter implements MatchFilter {
public boolean acceptMatch(CharSequence s, int start, int end) {
return (start == 0 || s.charAt(start-1) != '!');
}
}
```

Using the Transform Filter

The Transform Filter gives you more freedom to format your text strings by letting you modify the implicit URI generated by the link text. Decoupling the link text from the target URI gives you more freedom in how you display data strings to your users.

To use the Transform Filter, implement the transformUrl method in your Transform Filter. When Linkify fi nds a successful match, it calls transformUrl, passing in the RegEx pattern used and the default URI string it creates. You can modify the matched string, and return the URI as a target suitable to be "viewed" by another Android application.

The following TransformFilter implementation transforms the matched text into a lowercase URI:

```
class MyTransformFilter implements TransformFilter {
public String transformUrl(Matcher match, String url) {
return url.toLowerCase();
}
}
```

Returning Results from Activities

An Activity started using startActivity is independent of its parent and will not provide any feedback when it closes.

Alternatively, you can start an Activity as a sub-Activity that's inherently connected to its parent. Sub- Activities trigger an event handler within their parent Activity when they close. Sub-Activities are perfect for situations in which one Activity is providing data input (such as a user selecting an item from a list) for another.

Sub-Activities are created the same way as normal Activities and must also be registered in the application manifest. Any manifest-registered Activity can be opened as a sub-Activity.

Launching Sub-Activities

The startActivityForResult method works much like startActivity but with one important difference.

As well as the Intent used to determine which Activity to launch, you also pass in a *request code*. This value will be used later to uniquely identify the sub-Activity that has returned a result.

The skeleton code for launching a sub-Activity is shown below:

```
private static final int SHOW_SUBACTIVITY = 1;
Intent intent = new Intent(this, MyOtherActivity.class);
startActivityForResult(intent, SHOW_SUBACTIVITY);
```

As with regular Activities, sub-Activities can be started implicitly or explicitly. The following skeleton code uses an implicit Intent to launch a new sub-Activity to pick a contact:

```
private static final int PICK_CONTACT_SUBACTIVITY = 2;

Uri uri = Uri.parse("content://contacts/people");
Intent intent = new Intent(Intent.ACTION_PICK, uri);
startActivityForResult(intent, PICK_CONTACT_SUBACTIVITY);
```

Returning Results

When your sub-Activity is ready to close, call setResult before finish to return a result to the calling Activity.

The setResult method takes two parameters: the result code and result payload represented as an Intent. The result code is the "result" of running the sub-Activity — generally either Activity.RESULT_OK or Activity.RESULT_CANCELED. In some circumstances, you'll want to use your own response codes to handle application-specifi c choices; setResult supports any integer value.

The Intent returned as a result can include a URI to a piece of content (such as the contact, phone number, or media fi le) and a collection of Extras used to return additional information.

This next code snippet is taken from a sub-Activity's onCreate method and shows how an OK button and a Cancel button might return different results to the calling Activity:

```
Button okButton = (Button) findViewById(R.id.ok_button);
okButton.setOnClickListener(new View.OnClickListener() {
public void onClick(View view) {
Uri data = Uri.parse("content://horses/" + selected_horse_id);
Intent result = new Intent(null, data);
result.putExtra(IS_INPUT_CORRECT, inputCorrect);
result.putExtra(SELECTED_PISTOL, selectedPistol);

setResult(RESULT_OK, result);
finish();
}
});
Button cancelButton = (Button) findViewById(R.id.cancel_button);
cancelButton.setOnClickListener(new View.OnClickListener() {
public void onClick(View view) {
setResult(RESULT_CANCELED, null);
finish();
}
});
```

Handling Sub-Activity Results

When a sub-Activity closes, its parent Activity's onActivityResult event handler is fi red. Override this method to handle the results from the sub-Activities. The onActivityResult handler receives several parameters:

❑ **The Request Code** The request code that was used to launch the returning sub-Activity

❑ **A Result Code** The result code set by the sub-Activity to indicate its result. It can be any integer value, but typically will be either Activity.RESULT_OK or Activity.RESULT_CANCELLED. *If the sub-Activity closes abnormally or doesn't specify a result code before it closes, the result code is* Activity.RESULT_CANCELED.

❑ **Data** An Intent used to bundle any returned data. Depending on the purpose of the sub-Activity, it will typically include a URI that represents the particular piece of data selected from a list. Alternatively, or additionally, the sub-Activity can return extra information as primitive values using the "extras" mechanism.

The skeleton code for implementing the onActivityResult event handler within an Activity is shown below:

```
private static final int SHOW_SUB_ACTIVITY_ONE = 1;
private static final int SHOW_SUB_ACTIVITY_TWO = 2;
@Override
public void onActivityResult(int requestCode,
int resultCode,
Intent data) {
super.onActivityResult(requestCode, resultCode, data);
switch(requestCode) {
case (SHOW_SUB_ACTIVITY_ONE) : {
if (resultCode == Activity.RESULT_OK) {

Uri horse = data.getData();
boolean inputCorrect = data.getBooleanExtra(IS_INPUT_CORRECT,
false);
String selectedPistol = data.getStringExtra(SELECTED_PISTOL);
}
break;
}
case (SHOW_SUB_ACTIVITY_TWO) : {
if (resultCode == Activity.RESULT_OK) {
// TODO: Handle OK click.
}
break;
}
}

}
```

Native Android Actions

Native Android applications also use Intents to launch Activities and sub-Activities. The following noncomprehensive list shows some of the native actions available as static string constants in the Intent class. You can use these actions when creating implicit Intents to start Activities and sub-Activities within your own applications.

In the next section you will be introduced to Intent Filters, and you'll learn how to register your own Activities as handlers for these actions.

❑ ACTION_ANSWER Opens an Activity that handles incoming calls. Currently this is handled by the native phone dialer.

❑ ACTION_CALL Brings up a phone dialer and immediately initiates a call using the number supplied in the data URI. Generally, it's considered better form to use the Dial_Action if possible.

❑ ACTION_DELETE Starts an Activity that lets you delete the entry currently stored at the data URI location.

❑ ACTION_DIAL Brings up a dialer application with the number to dial prepopulated from the data URI. By default, this is handled by the native Android phone dialer. The dialer can normalize most number schemas; for example, tel:555-1234 and tel:(212) 555 1212 are both valid numbers.

❑ ACTION_EDIT Requests an Activity that can edit the data at the URI.

❑ ACTION_INSERT Opens an Activity capable of inserting new items into the cursor specifi ed in the data fi eld. When called as a sub-Activity, it should return a URI to the newly inserted item.

❑ ACTION_PICK Launches a sub-Activity that lets you pick an item from the URI data. When closed, it should return a URI to the item that was picked. The Activity launched depends on the data being picked; for example, passing content://contacts/people will invoke the native contacts list.

❑ ACTION_SEARCH Launches the UI for performing a search. Supply the search term as a string in the Intent's extras using the SearchManager.QUERY key.

❑ ACTION_SENDTO Launches an Activity to send a message to the contact specifi ed by the data URI.

❑ ACTION_SEND Launches an Activity that sends the specifi ed data (the recipient needs to be selected by the resolved Activity). Use setType to set the Intent's type as the transmitted data's mime type. The data itself should be stored as an extra using the key EXTRA_TEXT or EXTRA_STREAM depending on the type. In the case of e-mail, the native Android applications will also accept extras using the EXTRA_EMAIL, EXTRA_CC, EXTRA_BCC, and EXTRA_SUBJECT keys.

❑ ACTION_VIEW The most common generic action. View asks that the data supplied in the Intent's URI be viewed in the most reasonable manner. Different applications will handle view requests depending on the URI schema of the data supplied. Natively, http: addresses will open in the browser, tel: addresses will open the dialer to call the number, geo: addresses are displayed in the Google Maps application, and contact content will be displayed in the Contact Manager.

❑ ACTION_WEB_SEARCH Opens an activity that performs a Web search based on the text supplied in the data URI.

As well as these Activity actions, Android includes a large number of Broadcast actions that are used to create Intents that the system broadcasts to notify applications of events. These Broadcast actions are described later in this chapter.