Chapter 3 Android Components and Simple Examples

In this chapter, we present some simple examples, which are mostly adopted from the official Android tutorial web site where it explains in each example the meanings of statements and variables in the program but often misses the description of some steps that might be crucial for beginners to compile and run the program. Here we fill in those missing steps but may only describe the program context briefly.

3.1 GUI Example with Eclipse IDE

The first example is about **building a simple user interface** which can be found at

http://developer.android.com/training/basics/firstapp/building-ui.html.

We explain how to develop this user interface application using Eclipse IDE, which offers a lot of assistants to the user in the process. When an error occurs in the code, the IDE shows a small red circle at the left side of the problematic statement and underlines with red the fields that cause the error. The user can click on the red circle or the underlined field and Eclipse will present suggestions to fix the problem. Very often, an error may be caused by missing the import of a class; the user can fix it by clicking on the suggested import statement by Eclipse, which will then add the import statement to the program automatically.

We can build an Android graphical user interface (GUI) using a hierarchy of *View* and *ViewGroup* objects as shown in Figure 3-1 below. *View* objects in general consist of graphical widgets such as buttons and text fields while *ViewGroup* objects are *View* containers that are invisible, defining how the child views are laid out, such as in horizontal row or in a grid.

We can use XML to define a user interface (UI) with a hierarchy of UI elements.

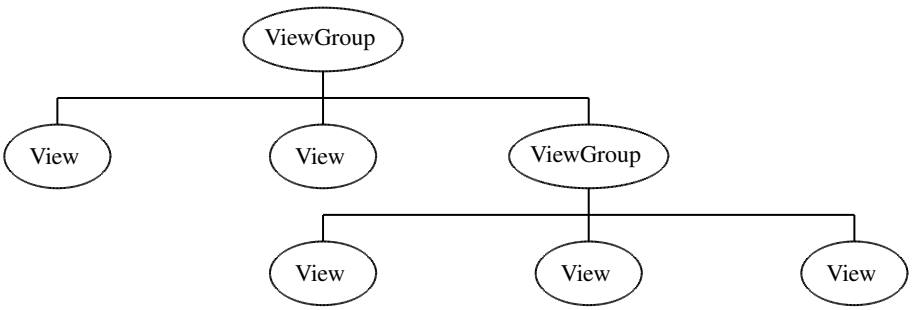


Figure 3-1. A Hierarchy of View and ViewGroup Objects

Suppose we call our application *SimpleGui* and we shall use the Android Linear Layout to present the interface. To start, we create the project using Eclipse IDE with:

File > **New** > **Android Application Project**

We can enter **SimpleGui**, **SimpleGui**, and **gui.simplegui** for *Application Name*, *Project Name* and *Package Name* respectively as shown in Figure 3-2 below.

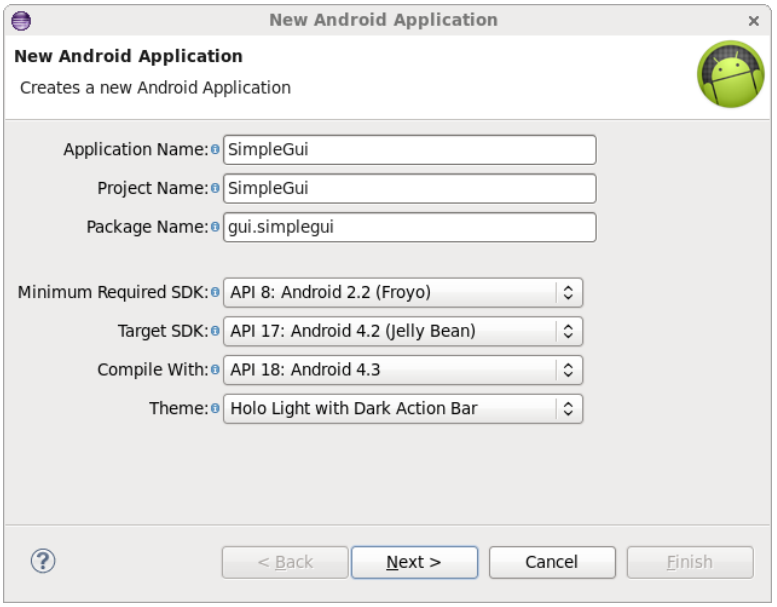


Figure 3-2 Creating SimpleGui Project

Then click **Next** > **Next** > **Next** > **Next**, where we have chosen to create a *Blank Activity*. In the *Blank Activity* panel, rather than using the default values, we enter **GuiMain** for *Activity Name* and *main* for *Layout Name*, and click **Finish** to create the **SimpleGui** project. The Eclipse IDe will show us the content of the layout file *main.xml* in the relative directory *res/layout* in graphical format, which is the default, as shown in Figure 3-3.

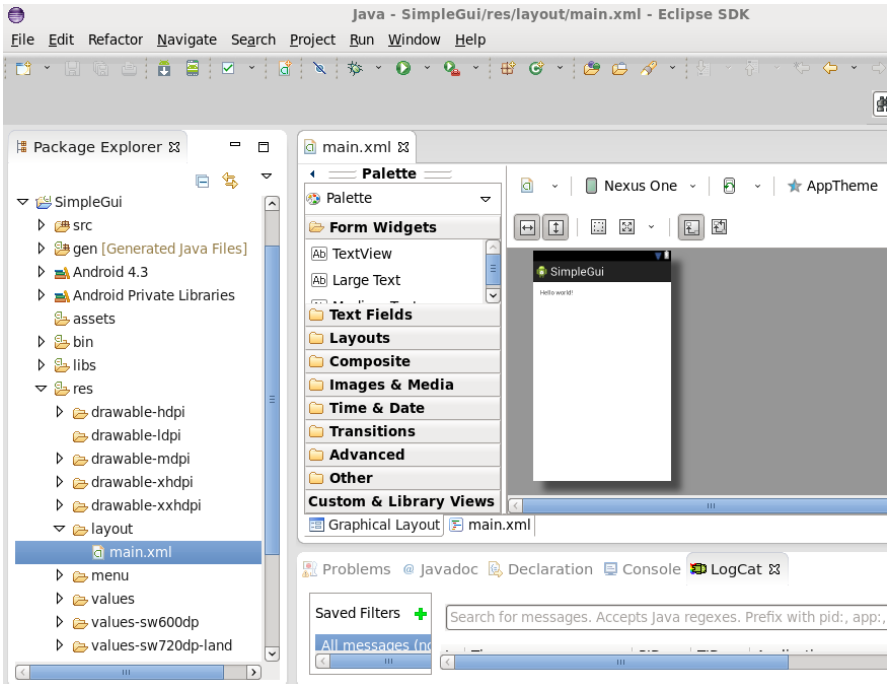


Figure 3-3 After Creating SimpleGui Project

To examine the file in text format, we have to click on the tab of *main.xml* shown at the bottom of the Eclipse editor.

3.1.1 Using a Linear Layout

The layout file *main.xml*, which is created by the *Blank Activity* template, defines a *RelativeLayout* and *TextView* child view. We want to change the layout to *LinearLayout*, which is a subclass of *ViewGroup*, laying out child views in either a vertical or horizontal orientation. So we modify the default *main.xml* file to the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal">
  <EditText android:id="@+id/message_box"
      android:layout_weight="1"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:hint="@string/message_box" />
  <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/send_button" />
</LinearLayout>
```

Two red error indicators will show up in the IDE editor at the two statements:

android:hint="@string/message_box"
android:text="@string/send_button"

This is because we have not defined the strings *message_box* and *send_button* in *string.xml*. To fix the errors we edit the file *strings.xml* in directory *res/values* to

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">SimpleGui</string>
  <string name="action_settings">Settings</string>
   <string name="message_box">Enter a message</string>
  <string name="send_button">Send</string>
</resources>
```

When we run the application, Android will display a GUI looked like Figure 3-4 below.

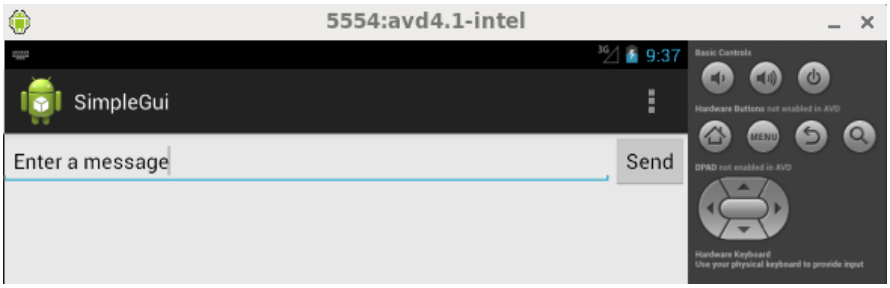


Figure 3-4 *SimpleGui* Display

The details of *layout* properties can be found at:

http://developer.android.com/guide/topics/ui/declaring-layout.html

What we have defined inside <LinearLayout> in the file *main.xml* above are a user-editable text field, <EditText> called *message_box*, and a <Button> called *send_button*.

The layout has been created in a way that both the **EditText** and **Button** widgets are only as big as necessary to fit their content, as shown in figure 3-4. The attributes and properties of the widgets are explained below.

<Edit Text> Attributes

android:id

This specifies a unique identifier for the view for further references. The symbol at (@) is required when specifying any resource object using XML. The plus sign (+) is also needed when we define a resource ID for the first time. In our example,

```
<EditText android:id="@+id/message_box"
```

message_box is the resource name and *id* is the resource type.

android:layout_width and **android:layout_height**

These attributes specify the width and height of the layout. The **wrap_content** value makes layout size to be as big as needed to fit the contents of the view. If the value **match_parent** is used, the *EditText* element will fill the screen, because it will match the size of the parent *LinearLayout*.

android:hint

This defines a default string for display. In our example, the string is given by *@string/message_box*, which specifies a string defined in the file *res/values/strings.xml*.

android:layout_weight

This defines the amount of remaining space each view should consume, relative to the amount taken up by other sibling views. The default value for a view is 0, implying that it won't fill any space by default.

<Button> Attributes

The **Button** attributes *layout_width* and *layout_height* of *main.xml* in our example are set to **wrap_content**, which makes the button widget to be as large as necessary to fit its content.

3.1.2 Responding to Clicking

For the button to respond to a user's clicking, we need to add the statement

android:onClick="transmitMessage"

to the <Button> element of the layout file *main.xml*. The attribute value "transmitMessage" of **android:onClick** is the method in the activity that the system calls when the button is clicked. We have to define this method in the class *GuiMain*. That is, we have to include this method in the file *GuiMain.java* in directory *src/gui/simplegui/* as shown below:

```
...
import android.view.View;
import android.content.Intent;
import android.widget.EditText;
```

```
...

public class GuiMain extends Activity
{
  public final static String EXTRA_MESSAGE = "gui.simplegui.MESSAGE";
  ...

  public void transmitMessage( View view )
  {
    Intent intent = new Intent(this, MessageActivity.class);
    EditText editText = (EditText) findViewById(R.id.message_box);
    String message = editText.getText().toString();
    intent.putExtra(EXTRA_MESSAGE, message);
    startActivity(intent);
  }
}
```

In the code, we have defined an *Intent* object, which provides late runtime binding between separate components such as two activities. In general, an *Intent* class is an abstract description of an operation to be performed, representing an app's "intent to do something". It is mostly used in launching activities, where it establishes a 'bridge' between activities. Detailed descriptions of the class can be found at

http://developer.android.com/reference/android/content/Intent.html

(Eclipse IDE allows us to look up a class conveniently; it may show a small green triangle on the scrolling bar at the statement of a new class. Clicking on the green triangle loads the information of the class.) In our example, an *Intent* is constructed to start an activity named *MessageActivity* using the constructor,

Intent intent = new Intent(this, MessageActivity.class);

which takes two parameters:

1. A *Context* object as its first parameter. The parameter *this* refers to the *GuiMain* class. Note that the *Activity* class is a subclass of the *Context* class.
2. The second parameter is an *Activity* that the *Intent* binds with the first.

Besides creating another activity, an *Intent* object can transmit data to the activity as well. In the **transmitMessage**() method, we use **findViewById**() to get the text from the **EditText** element and add it to the *Intent* object *intent*.

In order for the other activity to inquire the extra data, we have to define the key for the intent's extra as a public constant, which is the *String* constant **EXTRA_MESSAGE** defined in the *GuiMain* class above.

At this point, the Eclipse editor will indicate an error at the statement referencing *MessageActivity* as we have not defined this class. In the next section we discuss how to create an *Activity* class and define *MessageActivity*.

3.1.3 Creating a New Activity

To create a new *Activity*, click on the Eclipse IDE:

File > **New** > **Other..**

A window panel appears. Select the **Android** folder, choose **Android Activity**, and click **Next**. Then select **Blank Activity** and click **Next**, which presents a wizard like the one shown in Figure 3-5 below. Fill in the activity details:

Project: SimpleGui
Activity Name: MessageActivity
Layout Name: activity_message
Title: MessageActivity
Hierarchial Parent: gui.simplegui.GuiMain
Navigation Type: None

and click **Finish**, which creates the new **Activity**.

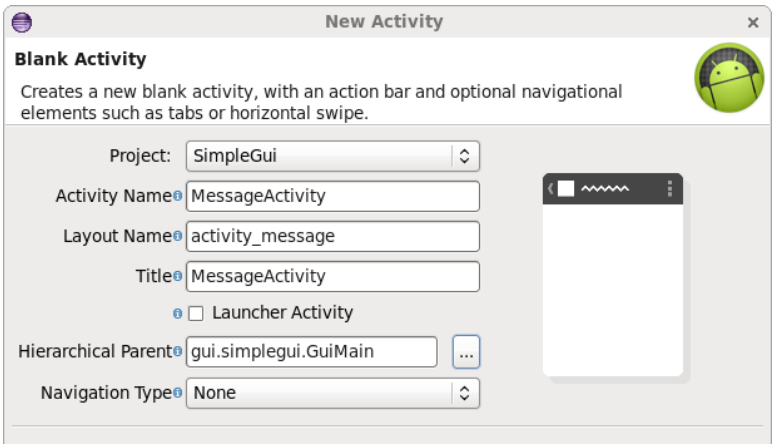


Figure 3-5 New Activity Wizard in Eclipse

The java file *src/gui/simplegui/MessageActivity.java* is created, consisting of a default template as follow:

```
package gui.simplegui;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.MenuItem;
import android.support.v4.app.NavUtils;
import android.annotation.TargetApi;
import android.os.Build;

public class MessageActivity extends Activity
{
  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_message);
    // Show the Up button in the action bar.
    setupActionBar();
  }

  /*
```

```
     * Set up the {@link android.app.ActionBar},if the API is available.
     */
    @TargetApi(Build.VERSION_CODES.HONEYCOMB)
    private void setupActionBar() {
      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        getActionBar().setDisplayHomeAsUpEnabled(true);
      }
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
      //Inflate the menu; this adds items to action bar if it is present.
      getMenuInflater().inflate(R.menu.message, menu);
      return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
     switch (item.getItemId()) {
     case android.R.id.home:
     //This ID represents the Home or Up button. In the case of this
     //activity, the Up button is shown. Use NavUtils to allow users
     //to navigate up one level in the application structure. For
     //more details, see the Navigation pattern on Android Design:
     /*
   http://developer.android.com/design/patterns/navigation.html#up-vs-back
      */
     NavUtils.navigateUpFromSameTask(this);
     return true;
    }
    return super.onOptionsItemSelected(item);
  }
}
```

We need to modify *MessageActivity* to the following so that it receives the message sent from *GuiMain* and displays it:

```
package gui.simplegui;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.MenuItem;
import android.support.v4.app.NavUtils;
import android.annotation.SuppressLint;
import android.annotation.TargetApi;
import android.content.Intent;
import android.os.Build;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

public class MessageActivity extends Activity
{
  @SuppressLint("NewApi")
  @Override
```

```
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_message);

    // Need running on Honeycomb or higher to use ActionBar APIs
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
      // Show the Up button in the action bar.
      getActionBar().setDisplayHomeAsUpEnabled(true);
    }
    // Get the message from the intent
    Intent intent = getIntent();
    String message = intent.getStringExtra(GuiMain.EXTRA_MESSAGE);

    // Create the text view
    TextView textView = new TextView(this);
    textView.setTextSize(40);
    textView.setText(message);

    // Set the text view as the activity layout
    setContentView(textView);
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    case android.R.id.home:
      NavUtils.navigateUpFromSameTask(this);
      return true;
    }
    return super.onOptionsItemSelected(item);
  }
}
```

You may see an error at the statement

@SuppressLint(”NewApi”)

To fix the error, you need to make sure that you use an API of 16 or higher and you may need to add to the program the import statement:

import android.annotation.SuppressLint;

When we run the app, type a message in the text field, and click the *Send* button, the message will show up on the second activity as shown in Figure 3-6 below.

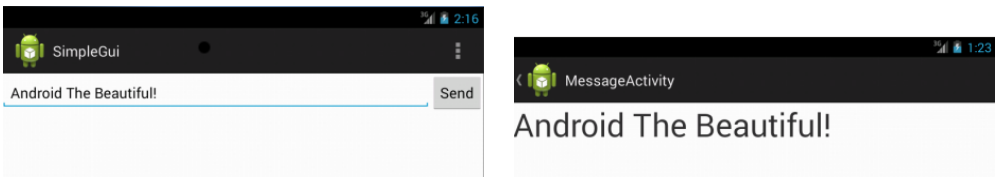


Figure 3-6 Both activities of SimpleGui Running on Android 4.3

3.2 GUI Example using Command Line

In this section we describe how to develop the above simple GUI app using command-line commands. We again assume that our workspace is in */workspace*. We will use the AVD called *SDcard*

that we created with Eclipse in the above example:

1. We first go to the workspace directory and create the subdirectory *simpleGui* with commands:

   $ cd /workspace/
   $ mkdir simpleGui

2. Create the project *simpleGui* by

   android create project –target 24 –name simpleGui –path ./simpleGui \
   –activity GuiMain –package gui.simplegui

3. Go into the project directory:

   $ cd simpleGui

4. Compile the project by

   $ ant debug

   to generate some default files.

5. Modify the activity program *GuiMain.java* using a text editor such as *vi*,

   $ vi src/gui/simplegui/GuiMain.java

   to the following:

```
package gui.simplegui;

import android.os.Bundle;
import android.app.Activity;
import android.content.Intent;
import android.view.Menu;
import android.view.View;
import android.widget.EditText;

public class GuiMain extends Activity {
  public final static String EXTRA_MESSAGE = "gui.simplegui.MESSAGE";
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu, adding items to the action bar
    getMenuInflater().inflate(R.menu.gui_main, menu);
    return true;
  }
  public void transmitMessage( View view )
  {
    Intent intent = new Intent(this, MessageActivity.class);
    EditText editText = (EditText) findViewById(R.id.message_box);
    String message = editText.getText().toString();
    intent.putExtra(GuiMain.EXTRA_MESSAGE, message);
    startActivity(intent);
  }
}
```

6. Create the other activity file using

$ vi src/gui/simplegui/MessageActivity.java

with the following code:

```
package gui.simplegui;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.MenuItem;
import android.support.v4.app.NavUtils;
import android.annotation.SuppressLint;
import android.annotation.TargetApi;
import android.content.Intent;
import android.os.Build;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

public class MessageActivity extends Activity {
  @SuppressLint("NewApi")
  @Override
  protected void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);
     setContentView(R.layout.activity_message);

      // Make sure we're running on Honeycomb or higher
      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
          // Show the Up button in the action bar.
          getActionBar().setDisplayHomeAsUpEnabled(true);
      }
      // Get the message from the intent
      Intent intent = getIntent();
      String message = intent.getStringExtra(GuiMain.EXTRA_MESSAGE);

      // Create the text view
      TextView textView = new TextView(this);
      textView.setTextSize(40);
      textView.setText(message);

      // Set the text view as the activity layout
      setContentView(textView);
  }
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
      switch (item.getItemId()) {
      case android.R.id.home:
        NavUtils.navigateUpFromSameTask(this);
        return true;
      }
      return super.onOptionsItemSelected(item);
  }
}
```

7. Modify *main.xml* with

   $ vi res/layout/main.xml

to the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="horizontal">
  <EditText android:id="@+id/message_box"
      android:layout_weight="1"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:hint="@string/message_box" />
  <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/send_button"
      android:onClick="transmitMessage"/>
</LinearLayout>
```

8. Create **activity_message.xml** using

   $ vi res/layout/activity_message.xml

   with the code:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:paddingBottom="@dimen/activity_vertical_margin"
  android:paddingLeft="@dimen/activity_horizontal_margin"
  android:paddingRight="@dimen/activity_horizontal_margin"
  android:paddingTop="@dimen/activity_vertical_margin"
  tools:context=".MessageActivity" >

  <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
</RelativeLayout>
```

9. Modify **strings.xml** with

   $ vi res/values/strings.xml

   to the following:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">SimpleGui</string>
  <string name="action_settings">Settings</string>
  <string name="message_box">Enter a message</string>
  <string name="send_button">Send</string>
  <string name="title_activity_message">MessageActivity</string>
</resources>
```

10. Create **dimens.xml** using

    $ vi res/values/dimens.xml

with the following code:

```
 <resources>
    <!-- Default screen margins, per the Android Design guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>
</resources>
```

11. Create **styles.xml** using

    $ vi res/values/styles.xml

    with the following code:

```
<resources>
  <style name="AppBaseTheme" parent="android:Theme.Light">
  </style>
  <!-- Application theme. -->
  <style name="AppTheme" parent="AppBaseTheme">
  </style>
</resources>
```

12. Modify **AndroidManifest.xml** with

    $ vi AndroidManifest.xml

    to the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="gui.simplegui"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="17" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
          android:name="gui.simplegui.GuiMain"
          android:label="@string/app_name" >
          <intent-filter>
             <action android:name="android.intent.action.MAIN" />
             <category android:name="android.intent.category.LAUNCHER" />
          </intent-filter>
        </activity>
        <activity
          android:name="gui.simplegui.MessageActivity"
          android:label="@string/title_activity_message"
          android:parentActivityName="gui.simplegui.GuiMain" >
          <meta-data
             android:name="android.support.PARENT_ACTIVITY"
             android:value="gui.simplegui.GuiMain" />
        </activity>
```

```
    </application>
</manifest>
```

13. Create a subdirectory *menu* in *res* by

    $ mkdir res/menu

    and create two files, *gui_main.xml* and *message.xml* inside this directory. Create the first one using command

    $ vi res/menu/gui_main.xml

    with code

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
   <item
     android:id="@+id/action_settings"
     android:orderInCategory="100"
     android:showAsAction="never"
     android:title="@string/action_settings"/>
</menu>
```

    and the second one using

    $ vi res/menu/message.xml

    with code

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
  <item
     android:id="@+id/action_settings"
     android:orderInCategory="100"
     android:showAsAction="never"
     android:title="@string/action_settings"/>
</menu>
```

14. To compile the programs, we need the android support library *android-support-v4.jar*. We can copy this file to our project library directory using a command similar to the following:

    $ cp ∼/Desktop/android-sdk-linux/extras/android/support/v4/android-support-v4.jar libs/ .

    (If the library file is not in the example path, you may locate it using the UNIX **locate** or **find** command.)

15. We start the emulator:

    $ emulator -avd SDcard &

16. We can now install the package:

    $ adb install bin/simpleGui-debug.apk

    When we run the app, we shall see activities similar to those shown in Figure 3-6.
    If you get an error message like *Unfortunately, simpleGui has stopped!* when you enter a message and click the **send** button, you might be using a newer API level for theme customization. In this case, you have to create a directory like "res/values-v*XX*" for the file "styles.xml" that specifies the application theme. For example, for API 14+ devices, the value of *XX* is 14; we first create the directory by

    $ mkdir res/values-v14

    and create the file *styles.xml* inside this directory using command

    $ vi res/values-v14/styles.xml

    with code

```
<resources>
  <style name="AppBaseTheme" parent="android:Theme.Holo.Light.DarkActionBar">
  </style>
</resources>
```

Recompile the app with the command “ant debug” and reintall it using the **adb** command with the “-r” option.
You may refer to the site

http://android-developers.blogspot.com/2012/01/holo-everywhere.html

for a discussion of the Holo theme used in *styles.xml*.

You can exit an activity by pressing the ‘ESC’ key.

3.3 Activity

3.3.1 Activity and Screen

In Android, an activity represents a screen as we have seen in the example of creating activities discussed above.

An application often consists of multiple activities that are loosely bound to each other. The application may create activities to generate different user-interface screens for various purposes. For example, the application can have a screen sending emails, a second screen for viewing a map and a third one for recording speech. Using activities, one can easily send messages from one screen to another. When we use Eclipse IDE to develop an application, the IDE generates a *main* activity which is a typical step in the development. When we run the application, the *main* activity is presented to us the first time we launch the application. When a new activity is started, the previous activity is stopped and is pushed onto a stack, which is last-in-first-out (LIFO). The stopped activity releases its large resource objects, such as network or database connections and reacquires them when it resumes actions. The state transitions are part of the activity lifecycle.

Android provides a few callback methods that an activity could receive because of a change in its state, regardless the nature of system – whether the system is creating the activity, stopping it, resuming it, or destroying it. Each callback provides the application a way to perform specific work that is appropriate to that state change.

Like a thread, an activity can be in one of several states in its life cycle: **active** (or running), **paused** or **stopped**. It transitions from one state to another in response to an event. In the transition,. it receives a call to a lifecycle method, which is defined in the Activity class

An **active** activity runs in the foreground, and “has the focus”. It is visible on the screen, allowing the user to interact with it. A **paused** activity is also visible on the screen but does not have the focus. A **stopped** activity is not visible on the screen.

A paused or stopped activity may be terminated by the system, when the system needs its memory for other tasks such as running another app.; a **stopped** actvitity has a higher priority of getting terminated.

For more details, one can refer to

http://developer.android.com/guide/components/activities.html

which is a link of the Android developers web site.

Readers can refer to the example presented in Section 3.2 and 3.3, where two activities (two screens) are created with one screen accepting an input message from the user and sending it to the other screen which accepts the message and displays it.

The steps of creating two activities called *main1* and *main2* representing two screens can be summarized as follows:

1. **XML Layouts**
   Create two XML layout files, *main1.xml*, and *main2.xml* in the directory *res/layout/* to represent screen 1 and screen 2 respectively:
   (a) File *res/layout/main1.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/linearLayout1"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I&apos;m Screen 1 (main1.xml)"
        android:textAppearance="?android:attr/textAppearanceLarge" />
    <Button
        android:id="@+id/button1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Click me to another screen" />
</LinearLayout>
```

   (b) File *res/layout/main2.xml*:
   Same as *main1.xml* except that the text *Screen 1 (main1.xml)* is substituted by *Screen 2 (main2.xml)*

3.3.2 A Simple Calculator

In this section, we present an example of a single activity, which is a calculator performing basic arithmetic operations including addition, subtraction, multiplication, and division. The user enters two numbers and clicks on an operation button. The result will be displayed along with the entered numbers and the operation. The code is simple and mostly self-explained. By studying this example, we can get familiar with the basic Android layout and program development. The following presents the procedures of creating and running this application using the Eclipse IDE.

1. **Create Project** *Calculator*:
   (a) Click **File** $>$ **New** $>$ **Project** $>$ **Android** $>$ **Android Application Project**
   (b) Specify the names of the project, the application and the package as *Calculator*, *Calculator*, and *example.calculator* respectively. Choose *API 11:Android 3.0(Honeycomb)* or later for the **Minimum Required SDK** entry. Then click **Next** $>$ **Next** $>$ **Next** $>$ **Next** to use the defaults of Eclipse. So the names of **Activity** and **Layout** are *MainActivity* and *activity_main* respectively. The **Navigation Type** is *None*. Then click **Finish** to create the project *Calculator*.

2. **Define Layout of** *MainActivity*:
   In this application, we have four different buttons, which are for the operations $+$, $-$, $*$, and $/$. Clicking on a button will cause the application perform the arithmetic operation on the two numbers entered in the edit-text spaces. So we define four *Button* objects within the main *LinearLayout* and two *EditText* object that allows a user to enter numbers. To accomplish these, we modify the file *res/layout/activity_main.xml* to the following code:

```
-------------------------------------------------------------------
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/linearLayout1"
    android:layout_marginLeft="12pt"
    android:layout_marginRight="12pt"
    android:layout_marginTop="4pt">
    <EditText
      android:layout_weight="1"
      android:layout_height="wrap_content"
      android:layout_marginRight="6pt"
      android:id="@+id/t1"
      android:layout_width="match_parent"
      android:inputType="numberDecimal">
    </EditText>
    <EditText
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:layout_marginLeft="6pt"
      android:id="@+id/t2"
      android:layout_width="match_parent"
      android:inputType="numberDecimal">
    </EditText>
  </LinearLayout>
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/linearLayout2"
    android:layout_marginTop="4pt"
    android:layout_marginLeft="6pt"
    android:layout_marginRight="6pt">
    <Button
      android:layout_height="wrap_content"
      android:layout_width="match_parent"
      android:layout_weight="1"
      android:text="+"
      android:textSize="10pt"
      android:id="@+id/plus">
    </Button>
    <Button
      android:layout_height="wrap_content"
      android:layout_width="match_parent"
      android:layout_weight="1"
      android:text="-"
      android:textSize="8pt"
      android:id="@+id/minus">
    </Button>
  </LinearLayout>

  <LinearLayout
    android:layout_width="match_parent"
```

```
  android:layout_height="wrap_content"
  android:id="@+id/linearLayout3"
  android:layout_marginTop="4pt"
  android:layout_marginLeft="6pt"
  android:layout_marginRight="6pt">

  <Button
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:layout_weight="1"
    android:text="*"
    android:textSize="10pt"
    android:id="@+id/multiply">
  </Button>
  <Button
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:layout_weight="1"
    android:text="/"
    android:textSize="10pt"
    android:id="@+id/divide">
  </Button>
  </LinearLayout>
  <TextView
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:layout_marginLeft="6pt"
    android:layout_marginRight="6pt"
    android:textSize="12pt"
    android:layout_marginTop="4pt"
    android:id="@+id/displayResult"
    android:gravity="center_horizontal">
  </TextView>
</LinearLayout>
-------------------------------------------------------------------------
```

3. Modify the main program *MainActivity.java* to the following code, which is mostly self-explained. One should study it carefully, trying to understand what each statement means.

```
-------------------------------------------------------------------------
package example.calculator;
import android.os.Bundle;
import android.app.Activity;
import android.content.DialogInterface;
import android.content.DialogInterface.OnClickListener;
import android.text.TextUtils;
import android.view.Menu;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends Activity
                      implements View.OnClickListener{
  EditText t1, t2;
```

```
Button plus, minus, multiply, divide;
TextView displayResult;
String oper = "";

/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  //find the EditText elements (defined in activity_main.xml)
  t1 = (EditText) findViewById(R.id.t1);
  t2 = (EditText) findViewById(R.id.t2);

  plus = (Button) findViewById(R.id.plus);
  minus = (Button) findViewById(R.id.minus);
  multiply = (Button) findViewById(R.id.multiply);
  divide = (Button) findViewById(R.id.divide);

  displayResult = (TextView) findViewById(R.id.displayResult);

  // set  listeners
  plus.setOnClickListener( this );
  minus.setOnClickListener( this);
  multiply.setOnClickListener( this);
  divide.setOnClickListener( this);
}

// @Override
public void onClick( View view ) {
  double num1 = 0;
  double num2 = 0;
  double result = 0;

  // check if the fields are empty
  if (TextUtils.isEmpty(t1.getText().toString())
      || TextUtils.isEmpty(t2.getText().toString())) {
    return;
  }

  // read EditText and fill variables with numbers
  num1 = Double.parseDouble(t1.getText().toString());
  num2 = Double.parseDouble(t2.getText().toString());

  // perform operations
  // save operator in oper for later use
  switch ( view.getId() ) {
  case R.id.plus:
    oper = "+";
    result = num1 + num2;
    break;
  case R.id.minus:
    oper = "-";
    result = num1 - num2;
    break;
```

```
        case R.id.multiply:
          oper = "*";
          result = num1 * num2;
          break;
        case R.id.divide:
          oper = "/";
          result = num1 / num2;
          break;
        default:
          break;
        }

        // form the output line
        displayResult.setText(num1 + " " + oper + " " + num2 +
" = " + result);
      }
}
```

4. **Run Application**:
   When we run the application, enter two numbers and click on an operation button, we should see an output screen similar to one shown on Figure 3-7

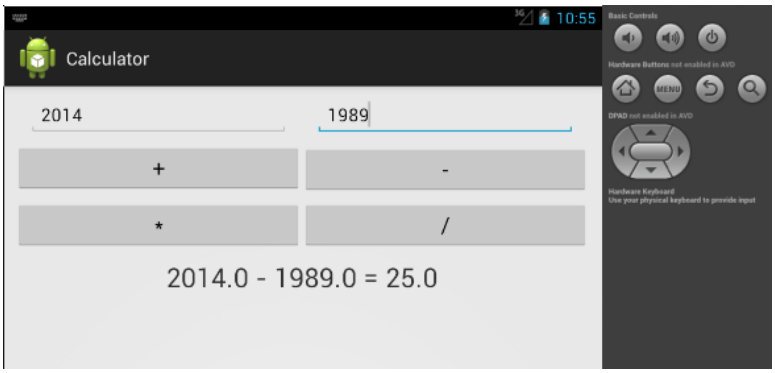


Figure 3-7 A Simple Calculator

3.4 Fragment

A *Fragment* is an independent component representing a behavior or a part of user interface in an Activity, but has its own lifecycle. A fragment may be considered as a modular section of an activity with functionality that can be easily reused within activities and layouts. It receives its own input events that we can easily add or remove while the activity is running. For example, a multi-pane user interface can be built by combining multiple fragments in a single activity and a fragment can be reused in multiple activities.

A fragment is always embedded in an activity; it must 'live' within an activity, running in the context of it. If the activity pauses, it pauses. If the activity terminates, it terminates. However, an activity may have multiple fragments and each fragment can live and operate independently from each other. We can dynamically or statically add fragments to an activity.

A fragment typically has its own user interface but it is also possible to construct a fragment without a user interface which is referred to as a headless fragment.

3.4.1 Why Fragments?

The primary reason for Android to introduce fragments since Android 3.0 (API level 11) is to make the user interface designs on large screens such as tablets, more dynamic and flexible. By dividing the layout of an activity into fragments, we can easily modify the activity's appearance at runtime and preserve the changes in a stack which is managed by the activity.

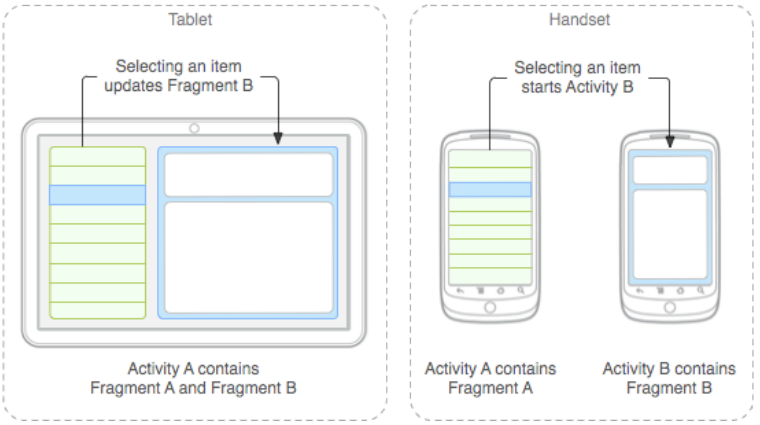


Figure 3-8 Fragments on Devices with Different Sizes
(Downloaded from *http://developer.android.com/guide/components/fragments.html*)

Fragments make an application easy to reuse components in different layouts. Figure 3-8 above, which is downloaded from the Android official site, shows an example of using fragments on devices with very different sizes.

The figure shows a typical example of the flexibility of using fragments, where we build single-pane layouts for a handset (phone) and multi-pane layouts for a tablets, which has a much larger screen size. On the tablet we see a list of details immediately on the same screen on the right hand side if we click on an item. On a smartphone the application presents a new detail screen. In general we should design a fragment as a modular and reusable activity component, which works independently from other fragments; it should be self-contained that defines its own layout.

3.4.2 Fragment Example

As an example of creating and working with fragments, we use the Eclipse IDE to create two fragments, *FragmentA* and *FragmentB* consisting of buttons, *buttonA* and *buttonB* respectively. When we click on *buttonA*, the display shows the *FragmentA* layout, and it shows *FragmentB* when *buttonB* is clicked. We name our project and application *FragmentDemo* with package *fragment.fragmentdemo*. As we need to extend the *Fragment* class, we need to use an Android API of version at least 11. The following are the steps of creating this application with Eclipse:

1. **Create Project** *FragmentDemo*:
   (a) Click **File** > **New** > **Project** > **Android** > **Android Application Project**
   (b) Specify the names of the project, the application and the package as *FragmentDemo*, *FragmentDemo*, and *fragment.fragmentdemo* respectively. Choose *API 11:Android 3.0(Honeycomb)* or later for the **Minimum Required SDK** entry. Then click **Next** > **Next** > **Next** > **Next** to use the defaults of Eclipse. So the names of **Activity** and **Layout** are *MainActivity* and *activity_main* respectively. The **Navigation Type** is *None*. Then click **Finish** to create the project *FramentDemo*.

2. **Define Layout of** *MainActivity*:
   In this application, we have two different buttons. Clicking on one displays a corresponding fragment interface below them. So we define two *Button* objects within the main *LinearLayout* and a *Fragment* object using the *fragment* tag. To accomplish these, we can modify the file *res/layout/activity_main.xml* to the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical" >

  <Button
    android:id="@+id/buttonA"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Fragment A"
    android:onClick="chooseFragment" />
  <Button
    android:id="@+id/buttonB"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:onClick="chooseFragment"
    android:text="Fragment B" />
  <fragment
    android:name="fragment.fragmentdemo.FragmentA"
    android:id="@+id/fragment_placeholder"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
</LinearLayout>
```

   In the layout, the <fragment> element defines an object of a *Fragment* class, in which **android:name** specifies the name of the class and **android:id** specifies the id of the *Fragment* object. *FragmentA* is the default *Fragment* but it can be replaced. So the id is just a place holder.

3. **Define Fragment Classes**:
   We create a simple class called *FragmentA* which displays a color background and a message:

   (a) In Eclipse IDE, click **File** > **New** > **Class**. Enter *fragment.fragmentdemo* and *FragmentA* for **Package:** and **Name:** respectively. Then click **Finish**.
   (b) Modify the file *src/fragment/fragmentdemo/FragmentA.java* to

```
package fragment.fragmentdemo;
import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class FragmentA extends Fragment {
  @Override
  public View onCreateView(LayoutInflater inflater,
```

```
  ViewGroup container, Bundle savedInstanceState) {
   //Inflate the layout for this fragment
   return inflater.inflate(R.layout.fragment_a,container,false);
  }
}
```

As we can see from the code, the class *FragmentA* extends the *Fragment* class and uses the **onCreateView** override method to create its user interface. The Android system calls this method, which returns a *View* component that is placed in the <fragment> element of the layout. The **inflate** method of *inflater* inflates a layout of an xml file and returns its view.

We smilarly create and define another class, *FragmentB*, which is almost identical to *FragmentA* except that we replace *fragment_a* by *fragment_b* in the first argument of **inflate**.

4. **Define Layout of Fragments**:
   Now we need to create and define the layouts for *FragmentA* and *FragmentB*:

   (a) In Eclipse, click **File** > **New** > Android XML File. The **New Android XML file** dialog shows up. Choose or enter *Layout*, *FragmentDemo* and *fragment_a* for **Resource Type**, **Project**, and **File** respectively. Then click **Finish**.

   (b) Modify the file *res/layout/fragment_a.xml* to the following to define the layout of *FragmentA*:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
         "http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:background="#ffffaa">
  <TextView
     android:id="@+id/textViewA"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     android:layout_weight="1"
     android:text="I am Fragment A"
     android:textSize="@dimen/font_size"
     android:textStyle="bold" />
</LinearLayout>
```

   This layout defines a background with color “ffffaa”, which has full red and green but less blue components. It displays the message “I am Fragment A” with bold font and text size specified in the file *dimens.xml*.

   (c) We similarly create another file, *fragment_b*.xml, that defines the layout for *FragmentB* that displays the message “I am Fragment B” with a different background color, which we set to “aaffff” that has less red component.

   (d) We can specify the dimensions of our layout in the file *dimens.xml*. In our example, we only need to specify the font size of text. To accomplish this, in Eclipse, click **File** > **New** > Android XML File. The **New Android XML file** dialog shows up. Choose or enter *Value*, *FragmentDemo* and *dimens* for **Resource Type**, **Project**, and **File** respectively. Then click **Finish**. Then modify the file *res/values/dimens.xml* to the following that specifies a large font size for text:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="font_size">64sp</dimen>
```

```
</resources>
```

5. **Write Code for** *MainActivity*:
   We need to write the code for the class *MainActivity*, which is the entry point of the application. We modify the file *src/fragment/fragmentdemo/MainActivity.java* to the following:

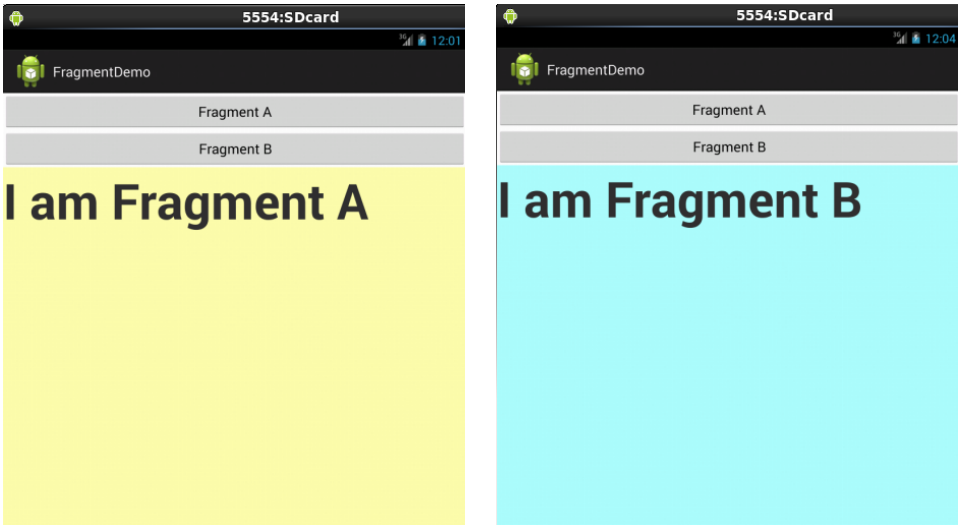
```
package fragment.fragmentdemo;
import android.os.Bundle;
import android.app.Activity;
import android.app.Fragment;
import android.app.FragmentManager;
import android.app.FragmentTransaction;
import android.view.View;
public class MainActivity extends Activity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
  }

  public void chooseFragment(View view) {
     Fragment frag;
     FragmentManager fragManager = getFragmentManager();
     FragmentTransaction fragTransaction =
                          fragManager.beginTransaction();
     if(view == findViewById(R.id.buttonA))
         frag = new FragmentA();
     else
         frag = new FragmentB();
     fragTransaction.replace(R.id.fragment_placeholder, frag);
     fragTransaction.commit();
  }
}
```

   As shown in the code, the class *FragmentTransaction* allows the application to add or replace the current fragment using the **add** or **replace** methods respectively. To commit the add or replace operation, we have to call the **commit** method as the code does. In this example, the current fragment is replaced by the chosen one when the user clicks on a button to choose the desired fragment. The first argument of the **replace** method is an integer identifier of the container whose fragment is to be replaced, and in our example, the id *fragment_placehoder* is specified in the <fragment> element in the activity_main layout.

6. **Run Application**:
   When we run the application, we will first see our default fragment, which is *FragmentA* as shown in Figure 3-9 (a) below. When we click on the *Fragment B* button, the layout of Figure 3-9 (b) will be displyed. Clicking on the *Fragment A* button will bring us back to the layout of Figure 3-9(a).



(a) (b)

Figure 3-9 Fragments in Same Activity

3.5 Service

3.5.1 Android Services

In Android, a *Service* is an activity running in the background. It does not have any direct user interface and thus is not bound to the activity life cycle. It is convenient to use services to carry out communications with remote components, downloading or sending data. A service can be also used to do time-consuming operations in the background, such as image and speech processing, 3D graphics modeling and data analysis. Each service class must have a corresponding *<service>* declaration in *AndroidManifest.xml* of its package. A service can be started with *Context*.**startService**() and *Context*.**bindService**(). It can essentially take two forms:

1. **Started**: A service can be *started* by an application component, such as an activity, which calls **startService**(). Once started, it can run in the background to perform operations such as downloading or uploading files indefinitely, even if the component that started it has terminated. However, the service should stop itself when the operation is done,
2. **Bound**: A service can be *bound* to an application component, which calls **bindService**(). Once bound, a service allows the component to interact with it like sending requests, getting results, or doing interprocess communication (IPC). A service can be bound to multiple components at the same time, and the service is destroyed when all of the components have unbound.

A service is not a separate process. It runs by default as a component of the process that contains the main thread of the application, unless otherwise specified. Normally, a service is not a thread. It is not a means itself to do work off of the main thread.

A service is run with a higher priority than an invisible or inactive activity. It may be terminated by the Android system because of insufficient system resources. We can configure the system to restart it once sufficient resources are available.

Asynchronous communication techniques are commonly used in a service to perform resource intensive tasks. Often a service runs as a new thread to process the data in the background and terminates when the processing is finished.

Permissions of access to a service is declared in its manifest <service> element. Correspondingly, other applications have to declare a <**uses-permission**> element in their own manifest in order to start, stop, or bind to the service.

Since GINGERBREAD, when we use *Context*.**startService(Intent)**, we can also set *Intent.FLAG_GRANT_READ_URI_PERMISSION* and/or *Intent.FLAG_GRANT_WRITE_URI_PERMISSION* on the Intent, which will grant the *Service* temporary access to the Intent's specific URIs.

An Android system provides and runs predefined services, which can be used by any Android application that has the appropriate permissions. These system services are normally accessed via the **getSystemService**() method of a specific Manager class.

An application can also define its own services, which are started from other components such as activities, broadcast receivers, and other services. Custom services can be local or remote.

To create a service, we must define a class that extends *Service* (or one of its subclasses), and we have to override a few methods including the following callback methods, which are most important:

1. **onStartCommand**(): This method is called when another component, such as an activity, request to start the service by calling **startService**().
2. **onBind**(): It is called when another component wants to bind to service to perform a task such as RPC, by calling **bindService**(). In the implementation, we must provide an interface for a client to communicate with the service by return an *IBinder*.
3. **onCreate**(): It is called only when the service is first created.
4. **onDestroy**(): It is called when the service is not longer in use. Our service should implement this to release any resources such as registered listneers and receivers.

Figure 3-10 below shows the life cycles of an unbound service and a bound service. A bounded service can be rebound by calling **onRebind**() after it has been stopped.

3.5.2 Local Services

It is common that a *Service* is run as a secondary component along with other parts of an application, in the same process as the rest of the components. By default, all components of an **.apk** run in the same process unless otherwise stated explicitly.

To create a service, we need to declare it in AndroidManifest.xml. As an example, we want to define a local service called *ServiceLocal*. We have to add in the file *res/AndroidManifest.xml* an xml element similar to the following:

```
<application
  .....
  <service
    android:name="ServiceLocal"
    android:icon="@drawable/icon"
    android:label="@string/service_name">
  </service>
</application>
```

We should put an icon image file named *icon.png* in the appropriate *drawable* directory (e.g. *res/drawable-mdpi*, and add a couple of entries in the file *res/values/strings.xml* like the following:

```
<resources>
  .....
  <string name="service_name">Service Local</string>
  <string name="local_service_started">Local Service Started</string>
  <string name="local_service_label">Local Service Label</string>
</resources>
```

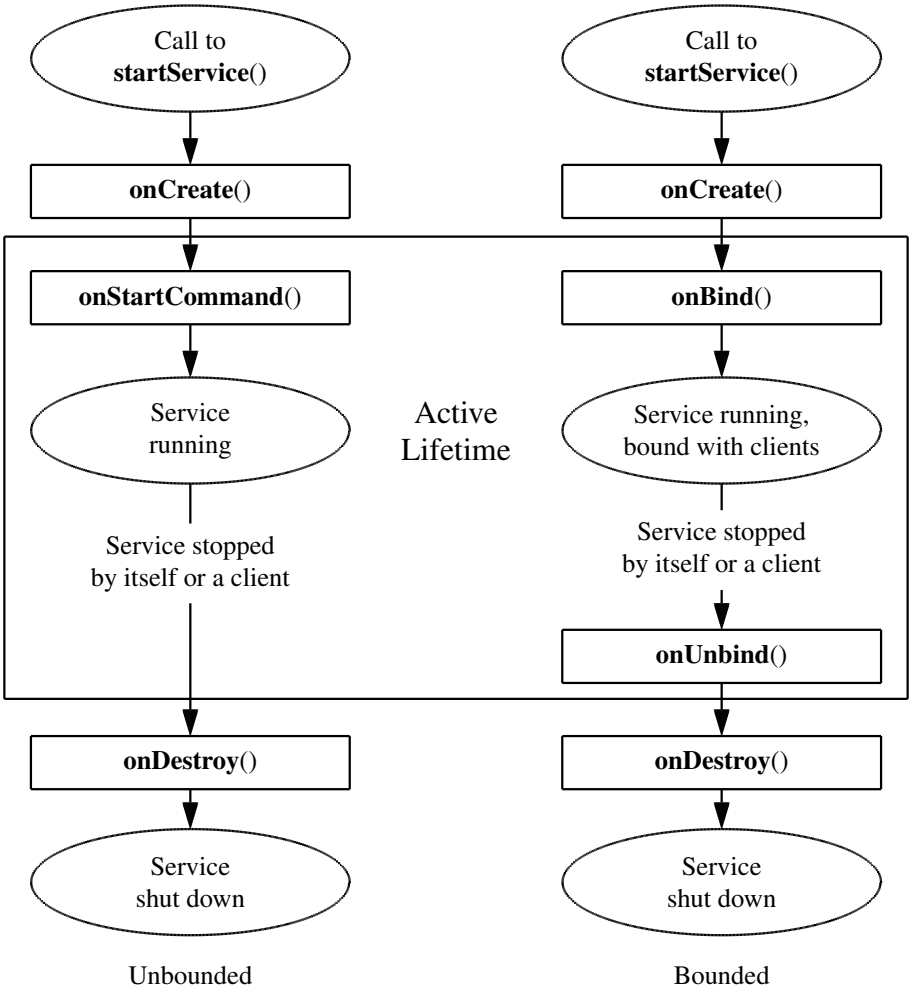


Figure 3-10 Life Cycles of Unbound and Bounded Services

To implement a service, we need to extend the *Service* class or one of its subclasses like the following:

```
public class ServiceLocal extends Service {

  @Override
  public int onStartCommand(Intent intent,int flags,int startId){
    ....
    return Service.START_NOT_STICKY;
  }

  // Class for clients to access.  Since this service is local, it always
  // runs in the same process as its clients.  So no need to deal with
  // Interprocess Communication (IPC).
  //
  public class LocalBinder extends Binder {
    ServiceLocal getService() {
      return ServiceLocal.this;
    }
  }
```

```
}
```

A service can be started by the **startService** method like the following:

```
Intent intent = new Intent(context, ServiceLocal.class);
// add extra data to the intent
intent.putExtra("Key", "Data to be used by service");
context.startService( intent );  //start service
```

Another way that we can start a service is to use the **bindService** method, which we will discuss later. It is a method of the class *ContextWrapper*, an extension of *Context*. The method allows us to communicate directly with the service.

If the service is not yet running when the **startService** method is called, a service object will be created and its **onCreate** method will be called. On the other hand, if the service is running when **startService** is called, its **onStartCommand** method is also called. The **onStartCommand** of a service is idempotent, allowing multiple calls without causing any harm to the tasks of the service.

The **onStartCommand**() method returns an **int**, which defines the restart behavior of the service if it is terminated by the Android platform. There are three common options of this returned **int**:

1. **START_STICKY**: The Service will be restarted automatically if for some reasons it is terminated. The *Intent* parameter passed to **onStartCommand**() should be null. This option is used for services that are independent of the *Intent* data, managing their own states by themselves.
2. **START_NOT_STICKY**: The Service will not be restarted automatically when it is terminated. It can be restarted by the **startService**() method.
3. **START_REDELIVER_INTENT**: This option is similar to **START_STICKY** except that the original *Intent* is redelivered to **onStartCommand**().

3.5.3 A Simple Local Service Example

We present in this section a simple example that illustrates how an activity binds to a local service. We implement a class called *ServiceLocal* to provide local service, which just returns a counter value. It extends the class *Service*, which is the base class for all services. The class *Toast* is used to flash messages on the screen. In general, a toast is a view containing a quick little message to be displayed on the screen.

In this appliction, the main activity presents a button on the screen. Whenever the button is pressed, a short message is displayed.

The following shows the steps of creating this application, assuming that we use Eclipse IDE to do the development. We call the project *ServiceLocal* with package name *service.servicelocal*.

1. Create project *ServiceLocal* from **File** > **New** > **Android Application** using names *ServiceLocal*, *ServiceLocal* and *service.serviclocal* for the names of **Application**, **Project** and **Package** respectively. Use defaults for other parameters, which generate the default main activity class file *MainActivity.java*, and the default layout file *activity_main.xml*.
2. Create class *ServiceLocal* from **File** > **New** > **Class**. Enter *ServiceLocal* for the class name. The Eclipse IDE generates the class file *ServiceLocal.java*. Modify this file to the following:

```
-----------------------------------------------------------------------
//ServiceLocal.java
package service.servicelocal;
import android.os.Binder;
import android.os.IBinder;
```

```
import android.util.Log;
import android.app.Service;
import android.widget.Toast;
import android.content.Intent;
import android.content.ComponentName;
import android.content.ServiceConnection;

public class ServiceLocal extends Service {
  private int counter = 0;

  public class LocalBinder extends Binder {
    ServiceLocal getService() {
      return ServiceLocal.this;
    }
  }

  int getCount() {
    // Some trivial task
    counter++;
    return counter;
  }

  @Override
  public int onStartCommand(Intent intent,int flags,int sId){
    Log.i("Service", "Received start id "+ sId + ": "+intent);
    //Return sticky so that this service continues to run until
    //it is explicitly stopped.
    return START_STICKY;
  }

  @Override
  public void onDestroy() {
    // Flash stopping message.
    Toast.makeText(this,"Service Stopped",Toast.LENGTH_SHORT).show();
  }
  @Override
  public IBinder onBind(Intent intent) {
    return iBinder;
  }
  // This object interacts with clients.
  private final IBinder iBinder = new LocalBinder();
}
```

---

3. Modify the main activity file *MainActivity.java* to the following. It defines the inner class *ServiceConnection* that establishes a connection to *ServiceLocal*. The method **bindServce**(), inherited from class *Context*, and called by **onCreate**(), binds the activity component to service *ServiceLocal*. This class also defines the **onClick** method, which is called when the button defined in *res/layout/activity_main.xml* is clicked; this method simply uses *Toast* to flash a brief message, indicating whether the service is bound successfully or not. If it is, the counter value of *ServiceLocal* is also displayed.

```
// MainActivity.java
package service.servicelocal;
```

```
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.os.IBinder;
import android.app.Activity;
import android.widget.Toast;
import android.content.Intent;
import android.content.Context;
import android.content.ComponentName;
import android.content.ServiceConnection;

public class MainActivity extends Activity {
  private ServiceLocal boundService;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Intent intent= new Intent(this, ServiceLocal.class);
    bindService(intent, connection, Context.BIND_AUTO_CREATE);
  }

  private ServiceConnection connection = new ServiceConnection(){
   @Override
   public void onServiceConnected(ComponentName name,
                                                  IBinder service){
     boundService =
                ((ServiceLocal.LocalBinder)service).getService();
     Toast.makeText(MainActivity.this, "Service connected!",
                                       Toast.LENGTH_SHORT).show();
   }

   @Override
   public void onServiceDisconnected(ComponentName name) {
     boundService = null;
     Toast.makeText(MainActivity.this, "Service disconnected",
                     Toast.LENGTH_SHORT).show();
   }
 };

 public void onClick(View view) {
   if ( boundService != null ) {
     Toast.makeText(this, "Service bound " +
              boundService.getCount(),Toast.LENGTH_SHORT).show();
   } else
     Toast.makeText(this, "Service not bound ",
                                       Toast.LENGTH_SHORT).show();
   }
}
```

4. Add the <service> element in the manifest description file *AndroidManifest.xml* like the following. It defines the service component.

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="service.servicelocal"
  android:versionCode="1"
  android:versionName="1.0" >
   ..........
  </activity>
  <service
     android:name="ServiceLocal">
  </service>
  </application>
</manifest>
```

5. Modify the layout description file *res/layout/activity_main.xml* to the following. The layout defines a button on the layout. When the button is clicked, the method **onClick**(), which is defined in *MainActivity.java* is called.

```
<LinearLayout xmlns:android =
              "http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:paddingBottom="@dimen/activity_vertical_margin"
   android:paddingLeft="@dimen/activity_horizontal_margin"
   android:paddingRight="@dimen/activity_horizontal_margin"
   android:paddingTop="@dimen/activity_vertical_margin"
   tools:context=".MainActivity" >
   <Button
       android:id="@+id/button1"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:onClick="onClick"
       android:text="Click Here" >
   </Button>
</LinearLayout>
```

When we run the application and click the displayed button, we will see a message flashed on the screen similar to the one shown in Figure 3-11 below.

3.5.4 Remote Services

We refer to a service that is in a different process from its binding client as a **remote service**. We have seen in the previous section how local services can be created and used. A local service runs in the same process as the application that started it and thus the life of it depends on the life of the said application. On the other hand, a remote service runs in its own process, which can be in another machine. Therefore, communicating with a remote service is more complicated as it involves inter-process communication (IPC), and devices may be connected by networks. Like the traditional RPC (remote procedure call), the object that is passed between two processes needs to be marshaled. For this purpose, Android provides the AIDL (Android Interface Definition Language) tool to handle data marshaling and communication.

An interface definition language (IDL) is a specification language used to describe a software component's interface in a language-independent way. This enables communication between soft-

ware components that do not use the same programming language, such as communication between components written in C++ and components written in Java. Remote procedure call (RPC) software commonly use IDLs are commonly used to bridge the components between two different systems.

To bind a remote service process, we need to create an AIDL file, which looks similar to a Java interface, but ends with the *.aidl* file extension. The service has to declare a service interface in an AIDL file, which is used by the AIDL tool to automatically create a java interface specified by it. The AIDL tool also generates a stub class that provides an abstract implementation of the service interface methods. We have to extend this stub class to provide the real implementation of the methods exposed through the interface. Also, the service clients need to invoke the **onBind**() method in order to connect to the service. One can refer to its official site at

http://developer.android.com/guide/components/aidl.html

for the detailed documentation of AIDL.

We will discuss remote services again in Chapter 8.

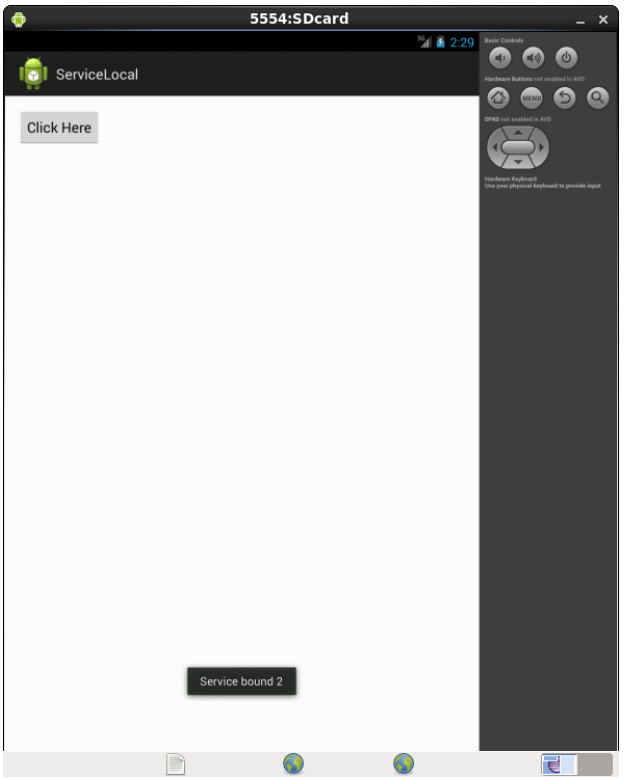


Figure 3-11 Local Service