MCA253 - Mobile Applications

Unit:1.2

Activities, Fragments, and Intents

Dr. Siddesha S MCA, M.Sc Tech (by Research) , Ph.D,
Asst. Professor,
Dept. of Computer Applications,
JSS Science and Technology University
Mysuru – 570 006

Understanding Activities

- An Android application can have zero or more *activities*.
- Typically, applications have one or more activities.
- The main purpose of an activity is to interact with the user.
- From the moment an activity appears on the screen to the moment it is hidden, it goes through a number of stages.
- These stages are known as an activity's *life cycle*.
- Understanding the life cycle of an activity is vital to ensuring that your application works correctly.

Create an activity

- To create an activity, you create a Java class that extends the Activity base class.

```
package com.jfdimarzio.chapter1helloworld;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
public class MainActivity extends AppCompatActivity {
@Override
protected void onCreate(Bundle savedInstanceState){
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
                                        }
 }
```

- Your activity class loads its user interface (UI) component using the XML file defined in your **res/layout** folder.
- In this example, you would load the UI from the main.xml file:

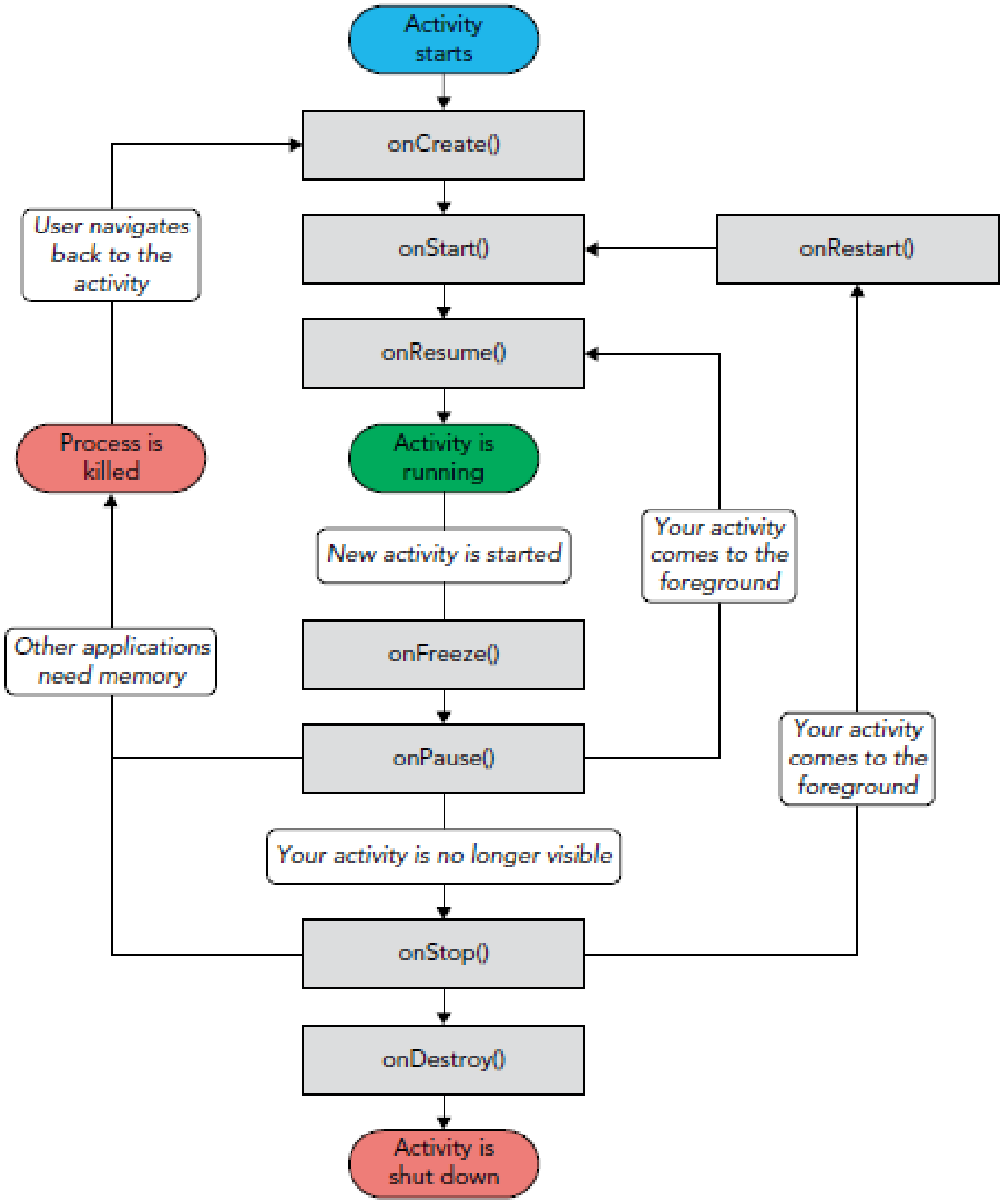
```
setContentView(R.layout.activity_main);
```

Create an activity

- Every activity of the application must be declared in AndroidManifest.xml file.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android
   package="com.jfdimarzio.chapter1helloworld">
  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
  <activity android:name=".MainActivity">
    <intent-filter>
         <action android:name="android.intent.action.MAIN" />
         <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>
 </application>
</manifest
```

Life cycle of an Activity


Activity starts
onCreate()
onStart()
onRestart()
User navigates back to the activity
onResume()
Process is killed
Activity is running
New activity is started
Your activity comes to the foreground
onFreeze()
Other applications need memory
onPause()
Your activity comes to the foreground
Your activity is no longer visible
onStop()
onDestroy()
Activity is shut down

Life cycle of an Activity

- The Activity base class defines a series of events that govern the life cycle of an activity.
- The Activity class defines the following events:
  - onCreate()—Called when the activity is first created
  - onStart()—Called when the activity becomes visible to the user
  - onResume()—Called when the activity starts interacting with the user
  - onPause()—Called when the current activity is being paused and the previous activity is being resumed
  - onStop()—Called when the activity is no longer visible to the user
  - onDestroy()—Called before the activity is destroyed by the system (either manually or by the system to conserve memory)
  - onRestart()—Called when the activity has been stopped and is restarting again

Life cycle of an Activity

- By default, the activity created contains the onCreate() event.
- Within this event handler is the code that helps to display the UI elements of screen.
- The best way to understand the various stages of an activity is to create a new project, implement the various events, and then subject the activity to various user interactions.

Understanding the Life Cycle of an Activity:

[Demonstration – ExampleActivity1]

```
package com.example.exampleactivity1;
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;
```

Understanding the Life Cycle of an Activity

```
public class MainActivity extends AppCompatActivity {
       String tag = "Life Cycle of an Activity";
       @Override
  protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
       Log.d(tag, "Inside OnCreate Event");
  }
  public void onStart(){
      super.onStart();
      Log.d(tag, "In the onStart() event");
  }
  public void onRestart() {
       super.onRestart();
       Log.d(tag, "In the onRestart() event");
  }
  public void onResume()
 {
       super.onResume();
       Log.d(tag, "In the onResume() event");
  }
```

Understanding the Life Cycle of an Activity

```
public void onPause()
{
     super.onPause();
     Log.d(tag, "In the onPause() event");
}
public void onStop()
{
     super.onStop();
     Log.d(tag, "In the onStop() event");
}
public void onDestroy()
{
     super.onDestroy();
     Log.d(tag, "In the onDestroy() event");
}
}
```

Applying Styles and Themes to an Activity

- By default, an activity is themed to the default Android theme.
- However, there has been a push in recent years to adopt a new theme known as Material.
- The Material theme has a much more modern and clean look to it.
- There are two versions of the Material theme available to Android developers: Material Light and Material Dark.
- Either of these themes can be applied from the AndroidManifest.xml.
- To apply one of the Material themes to an activity, simply modify the <Application> element in the AndroidManifest.xml file by changing the default android:theme attribute.
- Demonstration: **<ExampleActivity1>**

Hiding the Activity Title

- To hide the title of an activity if desired (such as when you just want to display a status update to the user).
- To do so, use the requestWindowFeature() method and pass it the Window
- .FEATURE_NO_TITLE constant, like this in onCreate() event of MainActivity.java file.

Syntax:

```
requestWindowFeature(Window.FEATURE_NO_TITLE);
```

- Now you need to change the theme in the AndroidManifest.xml to a theme that has no title bar

```
android:theme="@android:style/Theme.NoTitleBar">

<activity android:name=".MainActivity"

android:theme="@style/AppFullScreenTheme">
```

Displaying a Dialog Window

- There are times when you need to display a dialog window to get a confirmation from the user.
- In this case, you can override the onCreateDialog() protected method defined in the Activity base class to display a dialog window.

Demonstration Program Name: **AlertDialogDemo.**

Displaying a Progress Dialog

- One common UI feature in an Android device is the "Please wait" dialog that typically appear when an application is performing a long-running task.
- For example, the application might be logging in to a server before the user is allowed to use it, or it might be doing a calculation before displaying the result to the user.
- In such cases, it is helpful to display a dialog, known as a **progress dialog**, so that the user is kept in the loop.
- Demonstration Program Name: **ProgressDialog.**

Linking Activities Using Intents

- An Android application can contain zero or more activities.
- When a application has more than one activity, often need to navigate from one to another.
- In Android, one can navigate between activities through what is known as an intent.
- The best way to understand this very important but somewhat abstract concept is to experience it firsthand and see what it helps you achieve.

Demonstration Program Name: LinkActivityIntent.

Returning Results from an Intent

- The `startActivity()` method invokes another activity but does not return a result to the current activity.
- For example, you might have an activity that prompts the user for username and password.
- The information entered by the user in that activity needs to be passed back to the calling activity for further processing.
- If you need to pass data back from an activity, you should instead use the `startActivityForResult()` method.

Demonstration: `ResultFromActivity`

Passing Data Using an Intent Object

- Besides returning data from an activity, it is also common to pass data to an activity.
- For example, in the previous example, you might want to set some default text in the EditText view before the activity is displayed.
- In this case, you can use the Intent object to pass the data to the target activity.

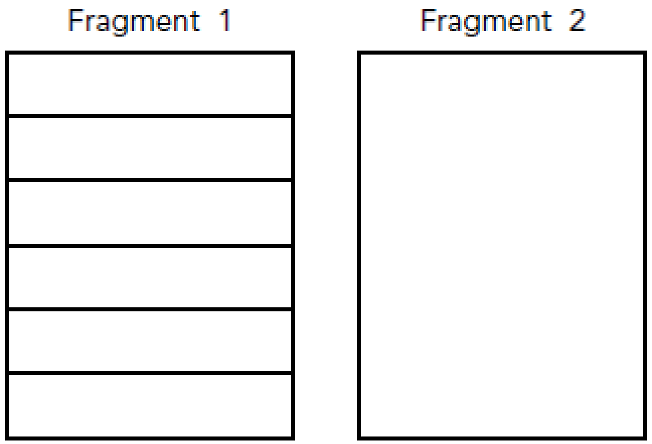
Demonstration: **`PassDataIntent`**

Fragments

- In a small-screen device (such as a smartphone), an activity typically fills the entire screen, displaying the various views that make up the user interface of an application.
- The activity is essentially a container for views.
- However, when an activity is displayed in a large-screen device, such as on a tablet, it is somewhat out of place.
- Because the screen is much bigger, all the views in an activity must be arranged to make full use of the increased space, resulting in complex changes to the view hierarchy.
- A better approach is to have “mini-activities,” each containing its own set of views. During runtime, an activity can contain one or more of these mini-activities, depending on the screen orientation in which the device is held.
- In Android 3.0 and later, these mini-activities are known as ***fragments***.

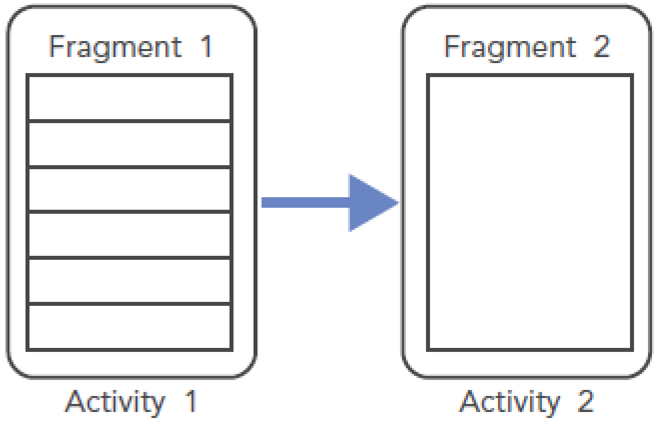
Fragments

- Think of a fragment as another form of activity. You create fragments to contain views, just like activities.
- Fragments are always embedded in an activity.



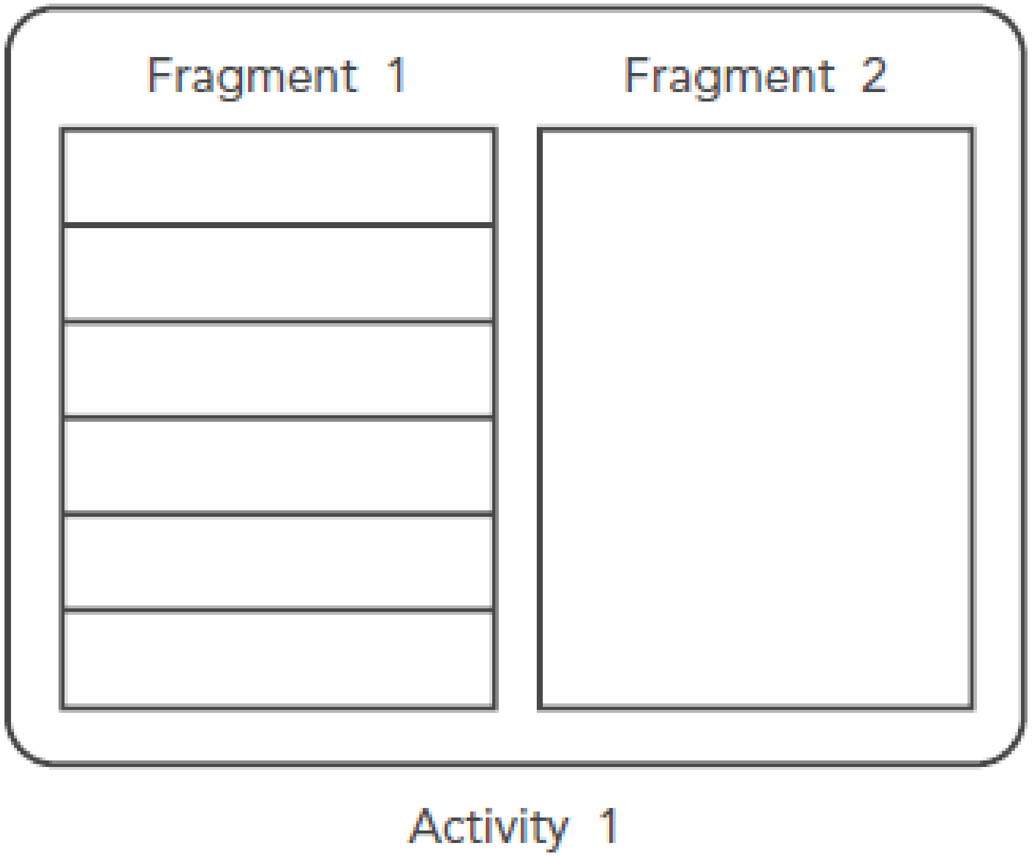
- Now imagine the application is running on an Android tablet (or on an Android smartphone) in portrait mode. In this case, Fragment 1 might be embedded in one activity, whereas Fragment 2 might be embedded in another activity. When users select an item in the list in Fragment 1, Activity 2 is started.

Fragments



- If the application is now displayed in a tablet in landscape mode, both fragments can be embedded within a single activity.
- It becomes apparent that fragments present a versatile way in which you can
- create the user interface of an Android application.
- Fragments form the atomic unit of your user interface, and they can be dynamically added (or removed) to activities in order to create the best user experience possible for the target device.

Fragments



Landscape view in case of Android Tablets

Demonstration: `Fragments`

Adding Fragments Dynamically

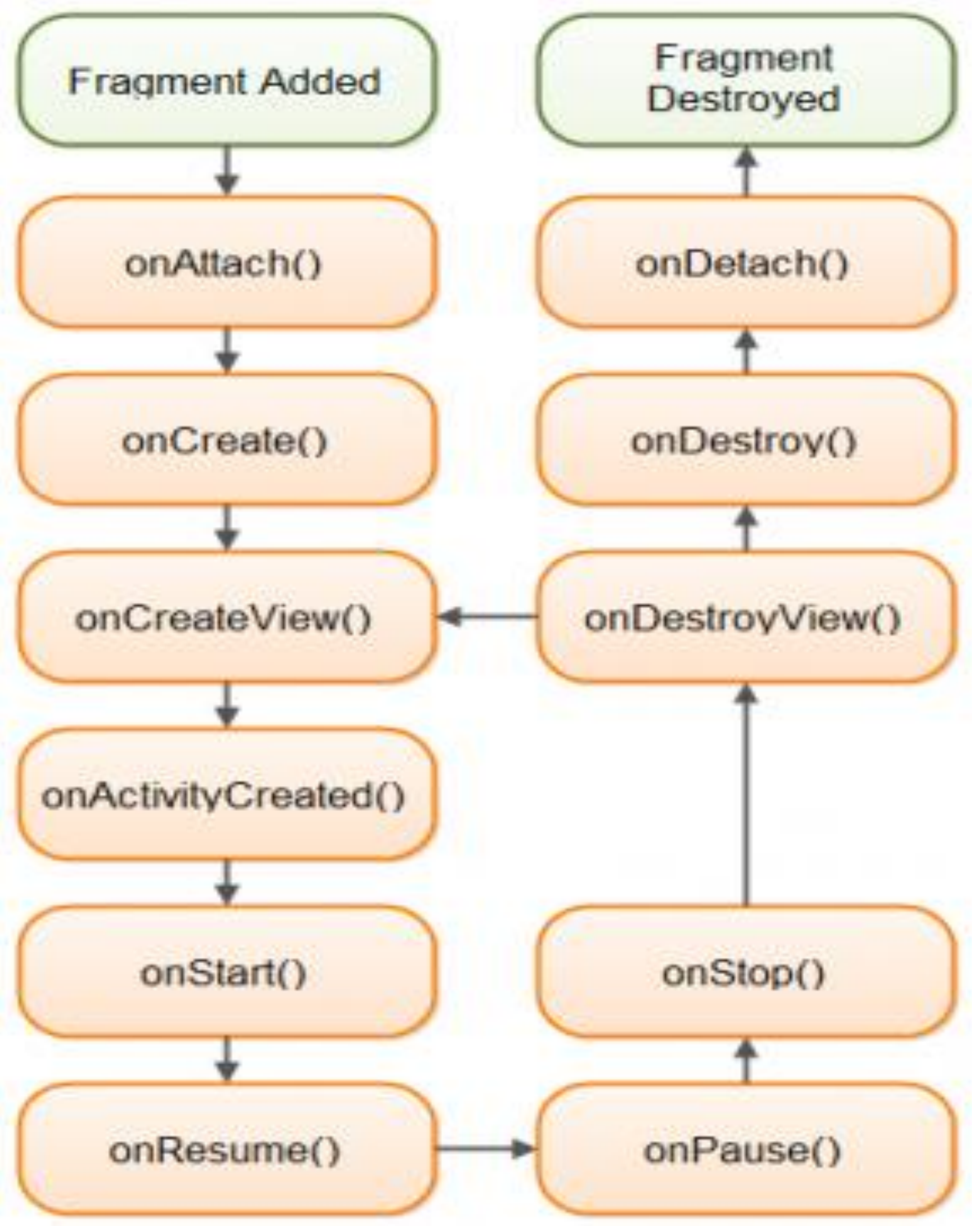
- Although fragments enable to compartmentalize the UI into various configurable parts, the real power of fragments is realized when they added dynamically to activities during runtime.
- In the previous section, how to add fragments to an activity by modifying the XML file during design time.
- In reality, it is much more useful if you create fragments and add them to activities during runtime.
- This enables to create a customizable user interface for your application.
- For example, if the application is running on a smartphone, fills an activity with a single fragment; if the application is running on a tablet, it might then fill the activity with two or more fragments, as the tablet has much more screen real estate compared to a smartphone.

Demonstration: `RunTimeFragments`

Life Cycle of a Fragment

- Like activities, fragments have their own life cycle.
- Understanding the life cycle of a fragment enables to properly save an instance of the fragment when it is destroyed, and restore it to its previous state when it is re-created.

Life Cycle of a Fragment



Life Cycle of a Fragment

- onAttach()—This method called first, To know that our fragment has been attached to an activity. We are passing the Activity that will host our fragment.
- onCreate()— This method called when a fragment instance initializes, just after the onAttach where fragment attaches to the host activity.
- onCreateView()— The method called when it’s time for the fragment to draw its user interface for the first time. To draw a UI for your fragment, you must return a View component from this method that is the root of your fragment’s layout. You can return null if the fragment does not provide a UI.
- onActivityCreated()—This method called when Activity completes its onCreate() method
- onStart()—This method called when a fragment is visible.

Life Cycle of a Fragment

- onResume()—This method called when a fragment is visible and allowing the user to interact with it. Fragment resumes only after activity resumes.

When fragment goes out off the screen:-

- onPause()—This method called when a fragment is not allowing the user to interact; the fragment will get change with other fragment or it gets removed from activity or fragment's activity called a pause.
- onStop()—This method called when the fragment is no longer visible; the fragment will get change with other fragment or it gets removed from activity or fragment's activity called stop.
- onDestroyView()— This method called when the view and related resources created in onCreateView() are removed from the activity's view hierarchy and destroyed.

Life Cycle of a Fragment

- onDestroy()— This method called when the fragment does its final clean up.
- onDetach()— This method called when the fragment is detached from its host activity.

How It Works:

- Like activities, fragments in Android also have their own life cycle. As you have seen, when a fragment is being created, it goes through the following states:
  - ➢ onAttach()
  - ➢ onCreate()
  - ➢ onCreateView()
  - ➢ onActivityCreated()
- When the fragment becomes visible, it goes through these states:
  - ➢ onStart()
  - ➢ onResume()

Life Cycle of a Fragment

- When the fragment goes into the background mode, it goes through these states:
  - onPause()
  - onStop()
- When the fragment is destroyed (when the activity in which it is currently hosted is destroyed), it goes through the following states:
  - onPause()
  - onStop()
  - onDestroyView()
  - onDestroy()
  - onDetach()

Life Cycle of a Fragment

- Like activities, you can restore an instance of a fragment using a Bundle object, in the following states:
    - onCreate()
    - onCreateView()
    - onActivityCreated()

- Most of the states experienced by a fragment are similar to those of activities. However, a few new states are specific to fragments:
    - onAttached()—Called when the fragment has been associated with the activity
    - onCreateView() Called to create the view for the fragment
    - onActivityCreated()—Called when the activity's onCreate() method has been returned
    - onDestroyView()—Called when the fragment's view is being removed
    - onDetach()—Called when the fragment is detached from the activity

Interactions Between Fragments

- An activity might contain one or more fragments working together to present a coherent UI to the user.
- In this case, it is important for fragments to communicate with one another and exchange data.
- For example, one fragment might contain a list of items (such as postings from an RSS feed).
- Also, when the user taps on an item in that fragment, details about the selected item might be displayed in another fragment.

RSS (RDF Site Summary or Really Simple Syndication) is a web **feed** that allows users and applications to access updates to websites in a standardized, computer-readable format. These **feeds** can, for example, allow a user to keep track of many different websites in a single news aggregator.

Demonstration: `InteractFragments`

Understanding the Intent Object

- So far, the Intent object is used to call other activities.
- First, we learnt that one can call another activity by passing its action to the constructor of an Intent object

```
startActivity(new Intent("com.test.SecondActivity"));
```

- The action (in this example "com.test.SecondActivity") is also known as the *component name*.
- This is used to identify the target activity/application that you want to invoke. You can also rewrite the component name by specifying the class name of the activity if it resides in your project, like this:

```
startActivity(new Intent(this, SecondActivity.class));
```

Understanding the Intent Object

- You can also create an Intent object by passing in an action constant and data, such as the following:

```
    Intent i = new
            Intent(android.content.Intent.ACTION_VIEW,
            Uri.parse("http://www.amazon.com"));
    startActivity(i);
```

- The action portion defines what you want to do, whereas the data portion contains the data for the target activity to act upon.
- You can also pass the data to the Intent object using the setData() method:

```
    Intent i = new
            Intent("android.intent.action.VIEW");
    i.setData(Uri.parse("http://www.amazon.com"));
```

Understanding the Intent Object

- In this example, you indicate that you want to view a web page with the specified URL.
- The Android OS will look for all activities that are able to satisfy your request. This process is known as *intent resolution*.
- For some intents, there is no need to specify the data.
- For example, to select a contact from the Contacts application, you specify the action and then indicate the MIME type using the setType() method:

```
Intent i = new
        Intent(android.content.Intent.ACTION_PICK);
i.setType(ContactsContract.Contacts.CONTENT_TYPE);
```

- The setType() method explicitly specifies the MIME data type to indicate the type of data to return.
- The MIME type for ContactsContract.Contacts.CONTENT_TYPE is "vnd.android.cursor.dir/contact".

Understanding the Intent Object

- Besides specifying the action, the data, and the type, an Intent object can also specify a category.
- A category groups activities into logical units so that Android can use those activities for further filtering.
- To summarize, an Intent object can contain the following information:
  - Action
  - Data
  - Type
  - Category

Using Intent Filters

- Earlier, you saw how an activity can invoke another activity using the Intent object.
- In order for other activities to invoke your activity, you need to specify the action and category within the <intent-filter> element in the AndroidManifest.xml file, like this:

```
<intent-filter >
    <action android:name="com.test.SecondActivity"/>
    <category android:name="android.intent.category.DEFAULT"/>
</intent-filter>
```

- This is a very simple example in which one activity calls another using the "com.test.SecondActivity" action.

Displaying Notifications

- So far, you have been using the Toast class to display messages to the user. While the Toast class is a handy way to show users alerts, it is not persistent.
- It flashes on the screen for a few seconds and then disappears.
- If it contains important information, users may easily miss it if they are not looking at the screen.
- For messages that are important, you should use a more persistent method.
- In this case, you should use the Notification Manager to display a persistent message at the top of the device, commonly known as the *status bar* (sometimes also referred to as the *notification bar*).