1. Explain the role of a Spinner and its adapters?
   (Explanation-2.5+steps for creation-2.5)

The Spinner view is an input control that provides you a drop-down list to select an item from a set of items. Generally, it shows a default value but you can select a new one from the drop-down list. The Spinner view needs a list of items to be displayed from an array of strings. You can do this by initializing a string array in the Java code or you can save the array of strings as a resource in the XML file. Generally, an Adapter class object provides the values to the drop down list of a Spinner.

Implementing a spinner

1. Create Spinner UI element in the XML layout
2. Define spinner choices in an array
3. Create Spinner and set onItemSelectedListener
4. Create an adapter with default spinner layouts
5. Attach the adapter to the spinner
6. Implement onItemSelectedListener method

1.b How to create Alert Dialog?
(Explanation-2.5+steps for creation-2.5)

Android has several types of dialogs.. Generally, dialogs are used where some decision or confirmation is to taken before proceeding ahead. A dialog box can be a date picker window, time picker window, a window showing the progress bar, a window asking for the confirmation with "Yes" and "No" buttons, etc. A Dialog is small window that prompts the user to a decision or enter additional information. Some of the dialogs are as follows:

1. **Alert Dialog:** It is a dialog box which is used to intimate the user to take the sudden action. Generally, it can has a title, up to three buttons, a custom layout, or a list of selectable items.
2. **Date Picker Dialog / Time Picker Dialog:** These dialog shows the DatePicker / TimePicker views as a dialog. User can select the appropriate date / time from the view as choose and pick method.

Alerts are urgent interruptions, requiring acknowledgement, that inform the user about a situation as it occurs, or an action *before* it occurs (as in discarding a draft). You can provide buttons in an alert to make a decision. For example, an alert dialog might require the user to click **Continue** after reading it, or give the user a choice to agree with an action by clicking a positive button (such as **OK** or **Accept**), or to disagree by clicking a negative button (such as **Cancel**).

Use the AlertDialog subclass of the Dialog class to show a standard dialog for an alert. The AlertDialog class allows you to build a variety of dialog designs. An alert dialog can have the following regions Title: A title is optional. Most alerts don’t need titles. If you can summarize a decision in a sentence or two by either asking a question (such as, “Discard draft?”) or making a statement related to the action buttons (such as, “Click OK to continue”), don’t bother with a title. Use a title if the situation is high-risk, such as the potential loss of connectivity, and the content area is occupied by a detailed message, a list, or custom layout.

1. Content area: The content area can display a message, a list, or other custom layout.
2. Action buttons: You should use no more than three action buttons in a dialog, and most have only two.

In order to make an alert dialog, you need to make an object of AlertDialogBuilder which an inner class of AlertDialog. Its syntax is given below

```
AlertDialog.Builder alertDialogBuilder = new AlertDialog.Builder(this);
```

Now you have to set the positive (yes) or negative (no) button using the object of the AlertDialogBuilder class. Its syntax is

```
alertDialogBuilder.setPositiveButton(
    "OK", newDialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
        // User clicked OK button.
    }
});
```

2.What are the different Screen navigation mechanisms?

2forms-+explanation-8, diagram+hierarchical-2

There are two forms of navigation

Temporal or back navigation

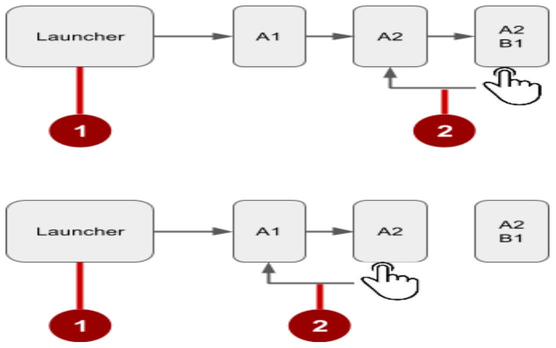
- provided by the device's back button

- ✕ controlled by the Android system's back stack

Ancestral or up navigation

- ✕ provided by the app's action bar
- ✕ controlled by defining parent-child relationships between activities in the Android manifest

1. Historys starts from Launcher
2. User clicks the Back button to navigate to the previous screens in reverse order –It will call finish() to finish current activity and stack next entry loading



3.Android system manages the back stack and Back button by overriding onBackPressed() method.

```
public void onBackPressed() {
   // Add the Back key handler here.
   return;
}
```

Hierarchical navigation patterns include

- ✕ **Parent screen**—Screen that enables navigation down to child screens, such as home screen and main activity
- ✕ **Collection sibling**—Screen enabling navigation to a collection of child screens, such as a list of headlines
- ✕ **Section sibling**—Screen with content, such as a story

The different types of Hierarchical navigation is

1. Descendant navigation
    a. Down from a parent screen to one of its children

       Eg:From a list of headlines to a story summary to a story
2. Ancestral navigation
    a. Up from a child or sibling screen to its parent
    b. Declare activity’s parent in Android manifest

```
<activity android:name=".OrderActivity"
  android:label="@string/title_activity_order"
  android:parentActivityName="com.example.android.
                       optionsmenuorderactivity.MainActivity">
  <meta-data
    android:name="android.support.PARENT_ACTIVITY"
    android:value=".MainActivity"/>
</activity>
```

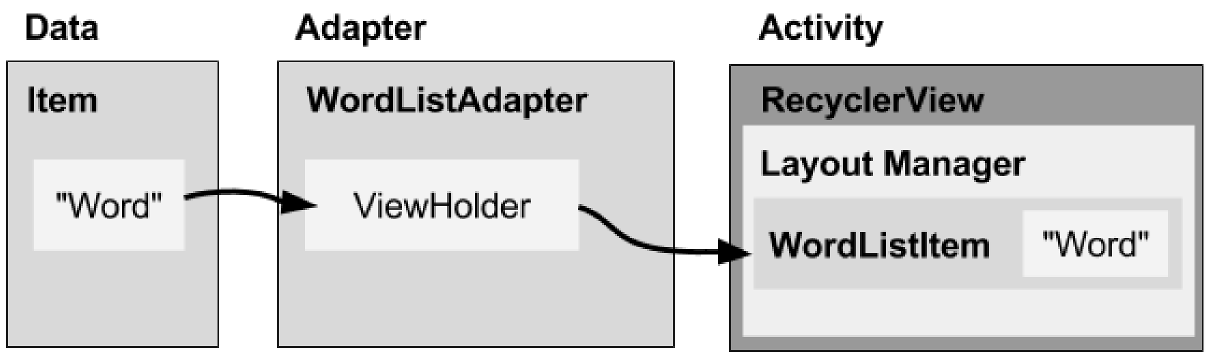
       Eg.From a story summary back to the headlines
3. Lateral navigation
    a. From one sibling to another sibling
    b. From a list of stories to a list in a different tab
    c. Swiping between tabbed views

Benefits of using tabs and swipes

- ✕ A single, initially-selected tab—users have access to content without further navigation
- ✕ Navigate between related screens without visiting parent

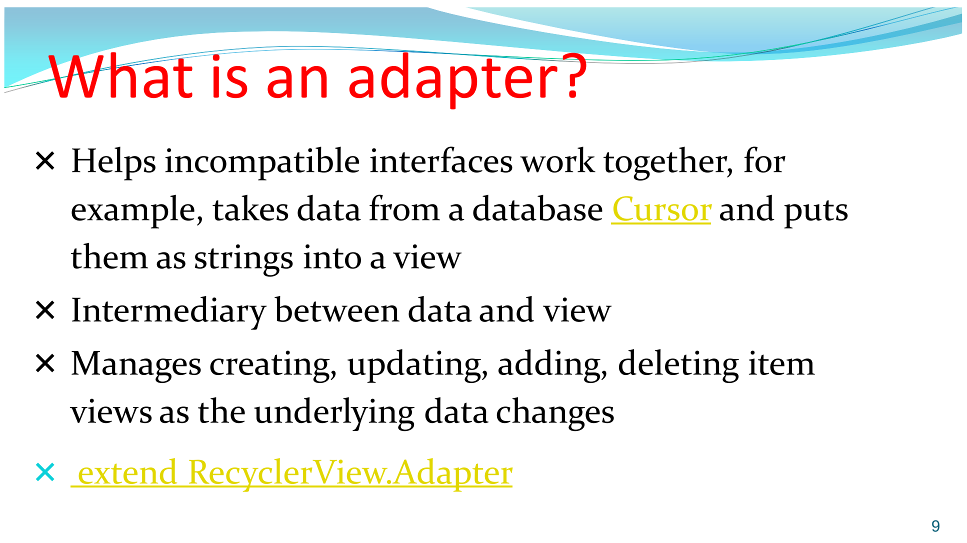
3.a.What are the different Recycler components?(6 components-5)

- **Data**
- **RecyclerView** scrolling list for list items—RecyclerView
- **Layout** for one item of data—XML file
- **Layout manage**r handles the organization of UI components in a view—Recyclerview.LayoutManager
- **Adapter** connects data to the RecyclerView—RecyclerView.Adapter
- **View holder** has view information for displaying one item—RecyclerView.ViewHolder



layout manager

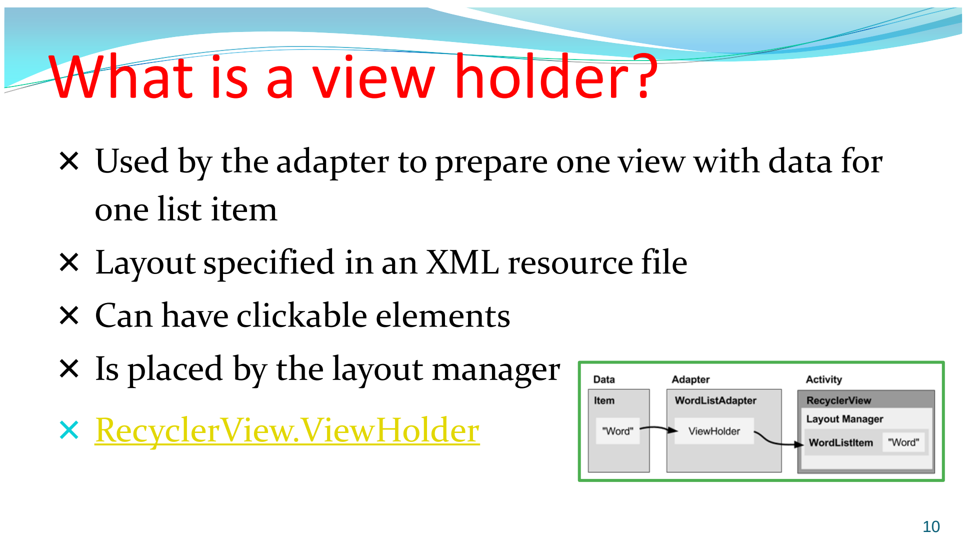
- All view groups have layout managers
- Positions item views inside a RecyclerView.
- Reuses item views that are no longer visible to the user
- Built-in layout managers include LinearLayoutManager, GridLayoutManager, and StaggeredGridLayoutManager
- For RecyclerView, extend RecyclerView.LayoutManager



What is an adapter?

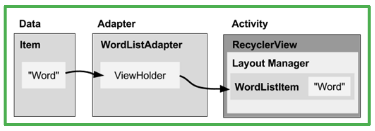
- Helps incompatible interfaces work together, for example, takes data from a database Cursor and puts them as strings into a view
- Intermediary between data and view
- Manages creating, updating, adding, deleting item views as the underlying data changes
- extend RecyclerView.Adapter

9



What is a view holder?

- Used by the adapter to prepare one view with data for one list item
- Layout specified in an XML resource file
- Can have clickable elements
- Is placed by the layout manager
- RecyclerView.ViewHolder



10

3.b.Drawable objects(any 5 components-5 marks)

- **Bitmap**: the simplest Drawable, a PNG or JPEG image.
- **Nine Patch**: an extension to the PNG format allows it to specify information about how to stretch it and place things inside of it.
- **Vector**: a drawable defined in an XML file as a set of points, lines, and curves along with its associated color information. This type of drawable can be scaled without loss of display quality.
- **Shape**: contains simple drawing commands instead of a raw bitmap, allowing it to resize better in some cases.
- **Layers**: a compound drawable, which draws multiple underlying drawables on top of each other.
- **States**: a compound drawable that selects one of a set of drawables based on its state.
- **Levels**: a compound drawable that selects one of a set of drawables based on its level.
- **Scale**: a compound drawable with a single child drawable, whose overall size is modified based on the current level.

4.a.Explain about AsyncTask and AsyncTaskLoader?

(Asynctask-3.5+ AsyncTaskLoader-3.5+limitations-3)

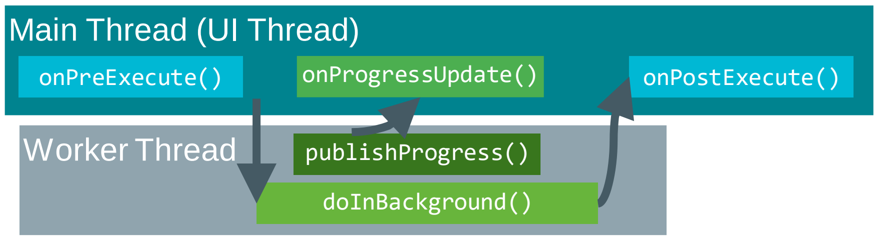
- Hardware updates screen every 16 milliseconds
- UI thread(main Thread) has 16 ms to do all its work

But there are operations which will take more time to finish

Eg;

1. Network operations
2. Long calculations
3. Downloading/uploading files
4. Processing images
5. Loading data etc

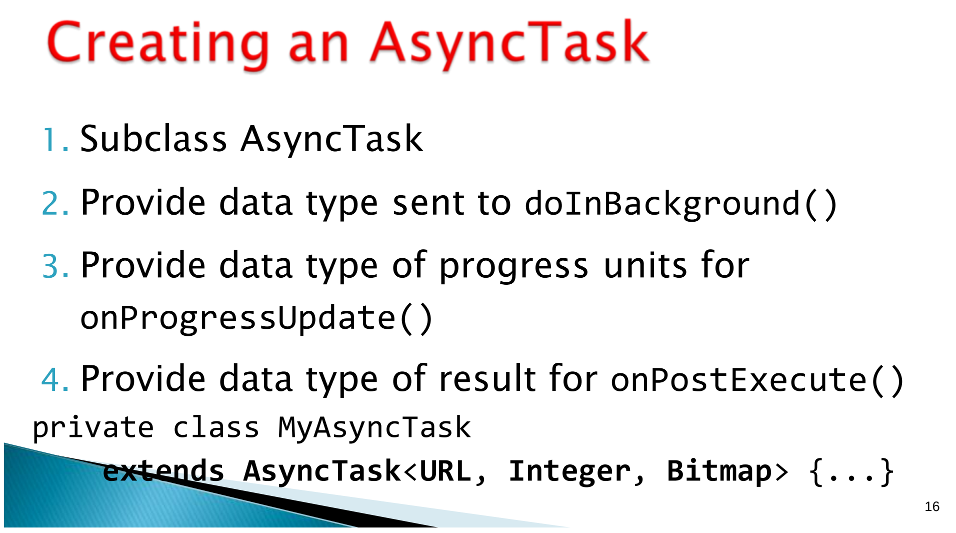
AsyncTask helper methods



15

- doInBackground()—runs on a background thread
    - All the work to happen in the background
- onPostExecute()—runs on main thread when work done
    - Process results
    - Publish results to the UI
- onPreExecute()

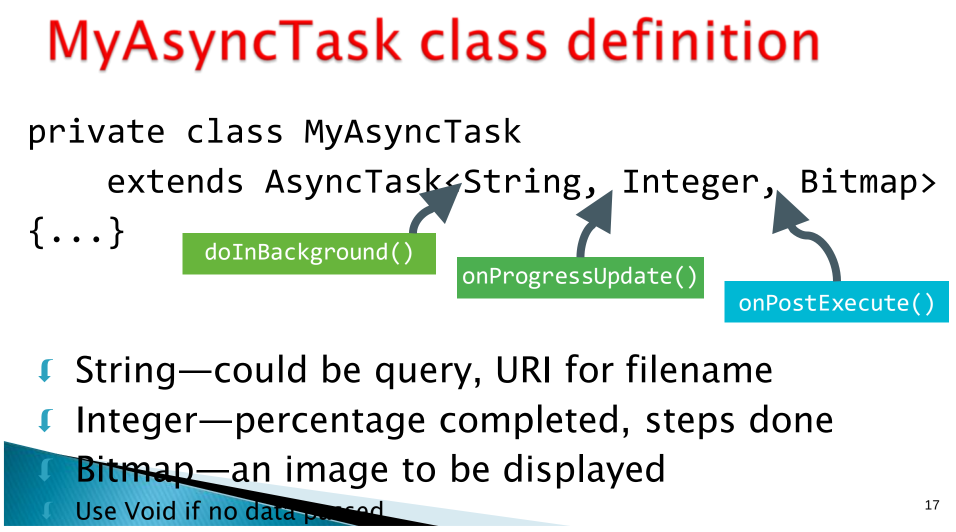
    - Runs on the main thread
    - Sets up the task
- onProgressUpdate()
    - Runs on the main thread
    - receives calls from publishProgress() from background thread



Creating an AsyncTask

1. Subclass AsyncTask
2. Provide data type sent to `doInBackground()`
3. Provide data type of progress units for `onProgressUpdate()`
4. Provide data type of result for `onPostExecute()`

```
private class MyAsyncTask
    extends AsyncTask<URL, Integer, Bitmap> {...}
```



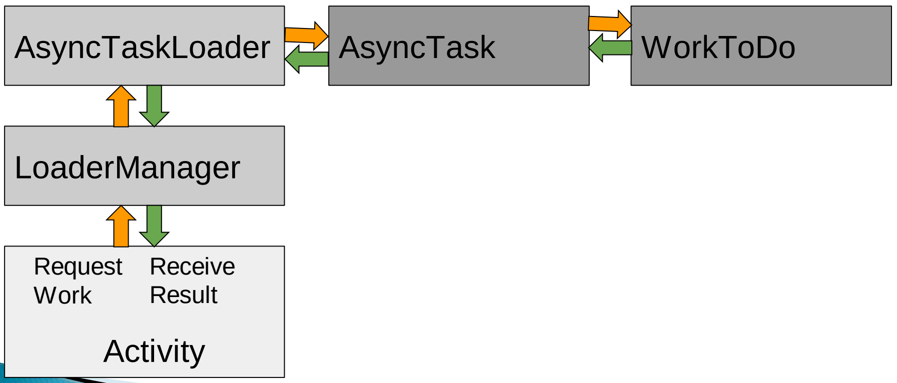
MyAsyncTask class definition

```
private class MyAsyncTask
    extends AsyncTask<String, Integer, Bitmap>
{...}
```

doInBackground()
onProgressUpdate()
onPostExecute()

- String—could be query, URI for filename
- Integer—percentage completed, steps done
- Bitmap—an image to be displayed
- Use Void if no data passed

AsyncTaskLoader Overview



AsyncTask $\longrightarrow$ AsyncTaskLoader

```
doInBackground()  → loadInBackground()
onPostExecute()   → onLoadFinished()
```

Get a loader with initLoader()

- Creates and starts a loader, or reuses an existing one, including its data
- Use restartLoader() to clear data in existing loader

-
  ```
  getLoaderManager().initLoader(Id, args, callback);
  getLoaderManager().initLoader(0, null, this);
  getSupportLoaderManager().initLoader(0, null, this);
  ```

onStartLoading()

When `restartLoader()` or `initLoader()` is called, the LoaderManager invokes the `onStartLoading()` callback

- Check for cached data
- Start observing the data source (if needed)
- Call `forceLoad()` to load the data if there are changes or no cached data

```
protected void onStartLoading() {  forceLoad();  }
```

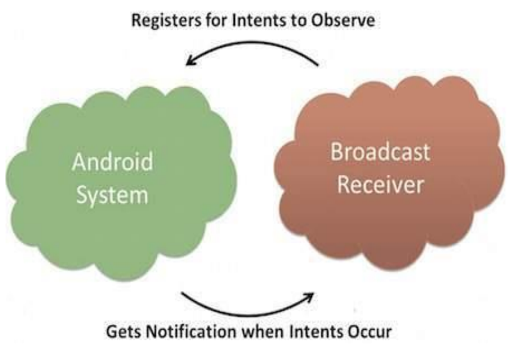
Implement loader callbacks in Activity

- `onCreateLoader()` — Create and return a new Loader for the given ID
- `onLoadFinished()` — Called when a previously created loader has finished its load
- `onLoaderReset()` — Called when a previously created loader is being reset making its data unavailable

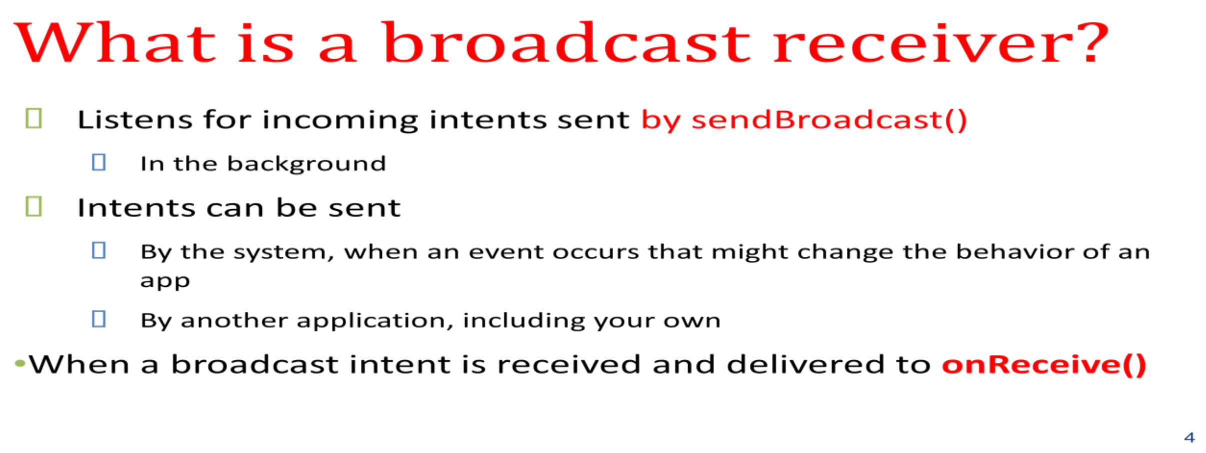
.What are the limitations of AsyncTask?

- When device configuration changes, Activity is destroyed
- AsyncTask cannot connect to Activity anymore
- New AsyncTask created for every config change
- Old AsyncTasks stay around
- App may run out of memory or crash

6. Explain about Broadcast intents and Broadcast receivers? (broadcast intent-5+receiver-5)


Registers for Intents to Observe
Android System
Broadcast Receiver
Gets Notification when Intents Occur

Broadcast Receivers simply respond to broadcast messages from other applications or from the system itself. These messages are sometime called events or intents. For example, applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use, so this is broadcast receiver who will intercept this communication and will initiate appropriate action.



What is a broadcast receiver?

- Listens for incoming intents sent by sendBroadcast()
  - In the background
- Intents can be sent
  - By the system, when an event occurs that might change the behavior of an app
  - By another application, including your own
- When a broadcast intent is received and delivered to **onReceive()**

4

There are following two important steps to make BroadcastReceiver works for the system broadcasted intents –

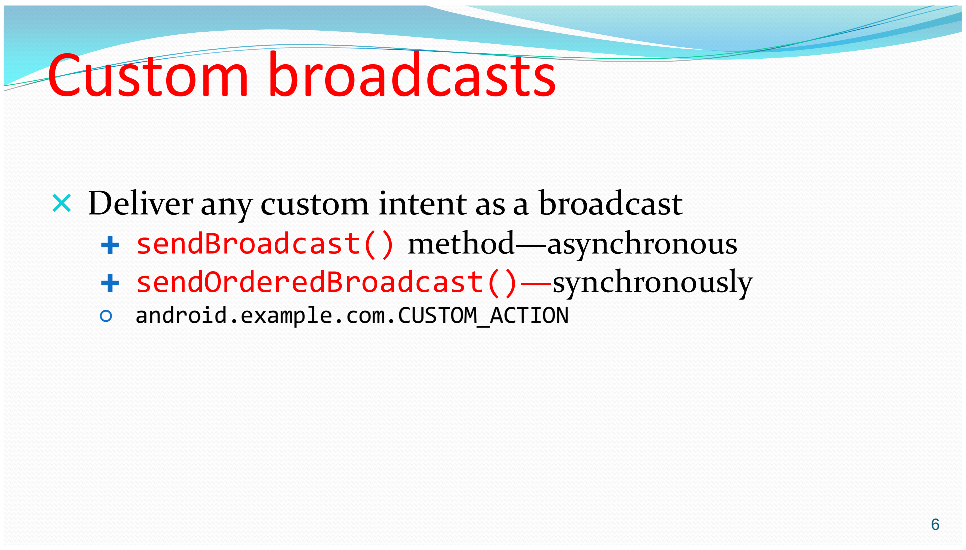
- Creating the Broadcast Receiver.

- Registering Broadcast Receiver

Creating the Broadcast Receiver

A broadcast receiver is implemented as a subclass of **BroadcastReceiver**class and overriding the onReceive() method where each message is received as a **Intent** object parameter.

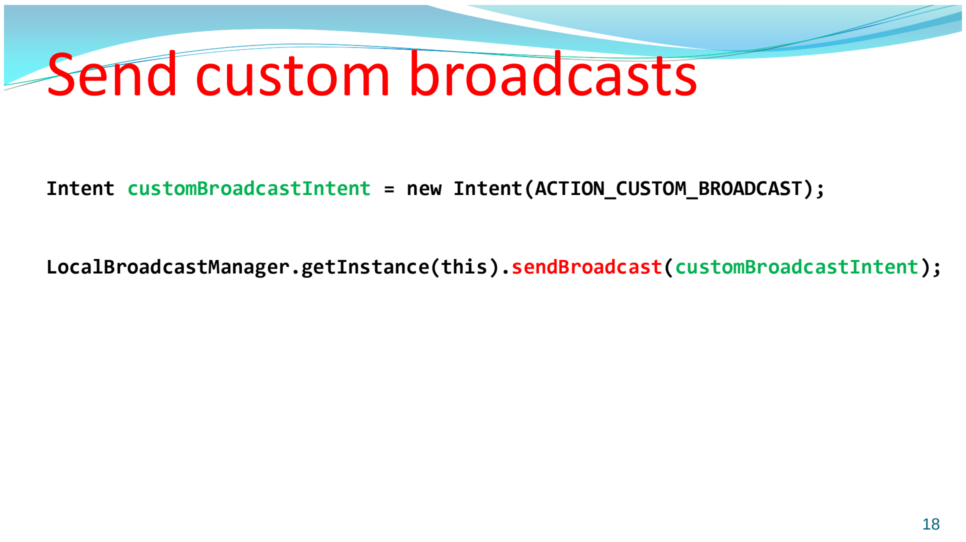
```
public class MyReceiver extends BroadcastReceiver {

   @Override

   public void onReceive(Context context, Intent intent) {

      Toast.makeText(context, "Intent Detected.", Toast.LENGTH_LONG).show();

   }

}
```



Custom broadcasts

- Deliver any custom intent as a broadcast
  - sendBroadcast() method—asynchronous
  - sendOrderedBroadcast()—synchronously
  - android.example.com.CUSTOM_ACTION

6



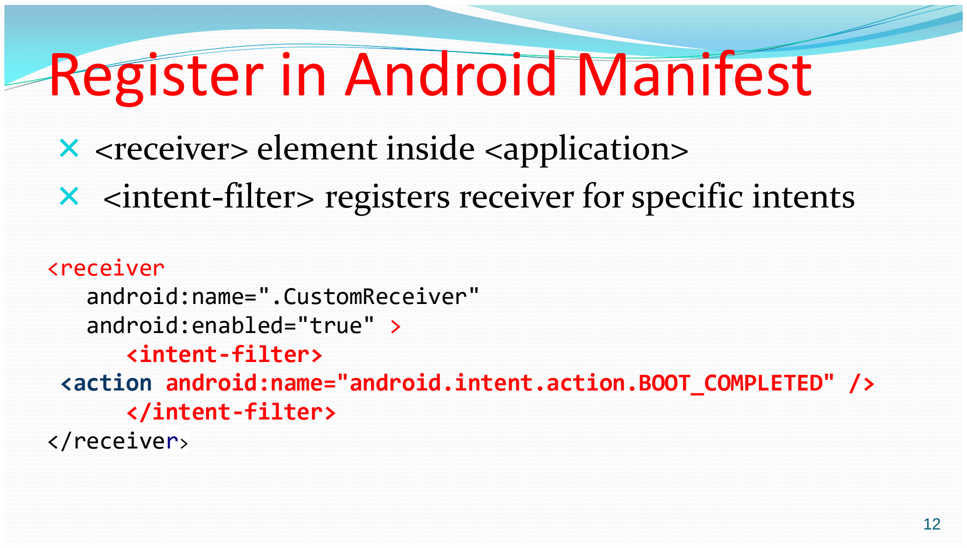
Send custom broadcasts

```
Intent customBroadcastIntent = new Intent(ACTION_CUSTOM_BROADCAST);

LocalBroadcastManager.getInstance(this).sendBroadcast(customBroadcastIntent);
```

Registering Broadcast Receiver

An application listens for specific broadcast intents by registering a broadcast receiver in *AndroidManifest.xml* file. Consider we are going to register *MyReceiver* for system generated event ACTION_BOOT_COMPLETED which is fired by the system once the Android system has completed the boot process.

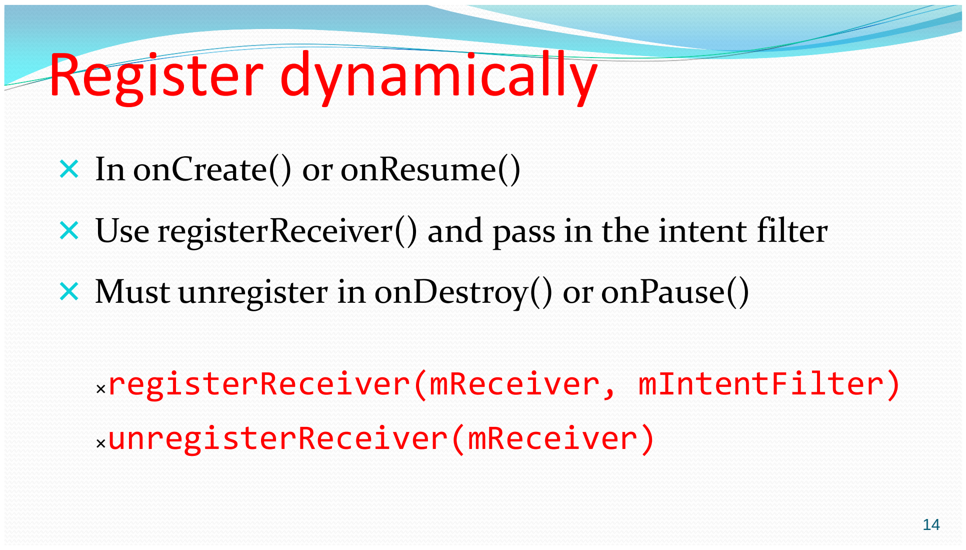
- Statically, in AndroidManifest
- Dynamically, with registerReceiver()



Register in Android Manifest

- <receiver> element inside <application>
- <intent-filter> registers receiver for specific intents

```
<receiver
   android:name=".CustomReceiver"
   android:enabled="true" >
      <intent-filter>
 <action android:name="android.intent.action.BOOT_COMPLETED" />
      </intent-filter>
</receiver>
```



Register dynamically

- In onCreate() or onResume()
- Use registerReceiver() and pass in the intent filter
- Must unregister in onDestroy() or onPause()

```
registerReceiver(mReceiver, mIntentFilter)
unregisterReceiver(mReceiver)
```

14

6.What is a service? Explain Service lifecycle?

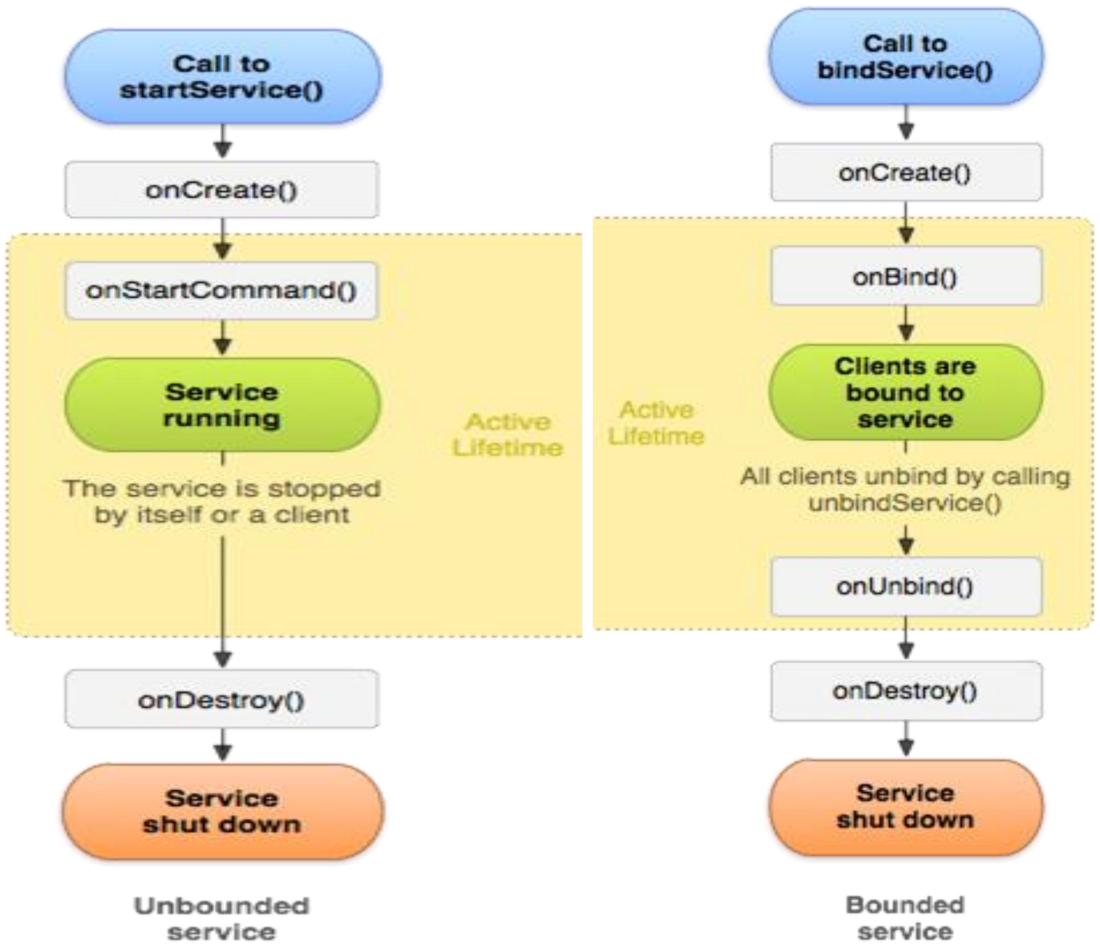
A **service** is a component that runs in the background to perform long-running operations without needing to interact with the user and it works even if application is destroyed. A service can essentially take two states –

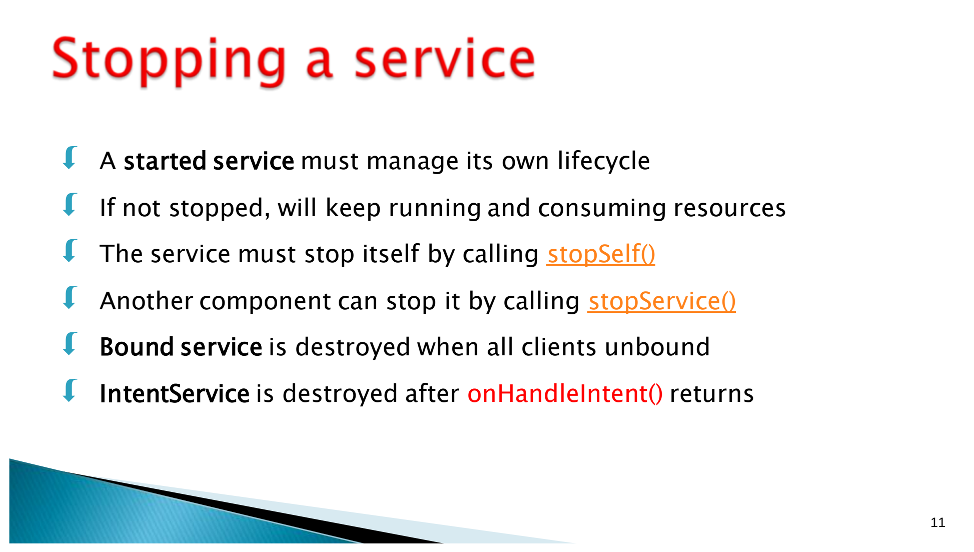
Sr.No.	State & Description
1	**Started** A service is **started** when an application component, such as an activity, starts it by calling *startService()*. Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.
2	**Bound** A service is **bound** when an application component binds to it by calling *bindService()*. A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC).

A service has life cycle callback methods that you can implement to monitor changes in the service's state and you can perform work at the appropriate stage. The following diagram on the left shows the life cycle when the service is created with startService() and the diagram on the right shows the life cycle when the service is created with bindService():

Characteristics of services

- Started with an Intent
- Can stay running when user switches applications
- Lifecycle—which you must manage
- Other apps can use the service—manage permissions
- Runs in the main thread of its hosting process





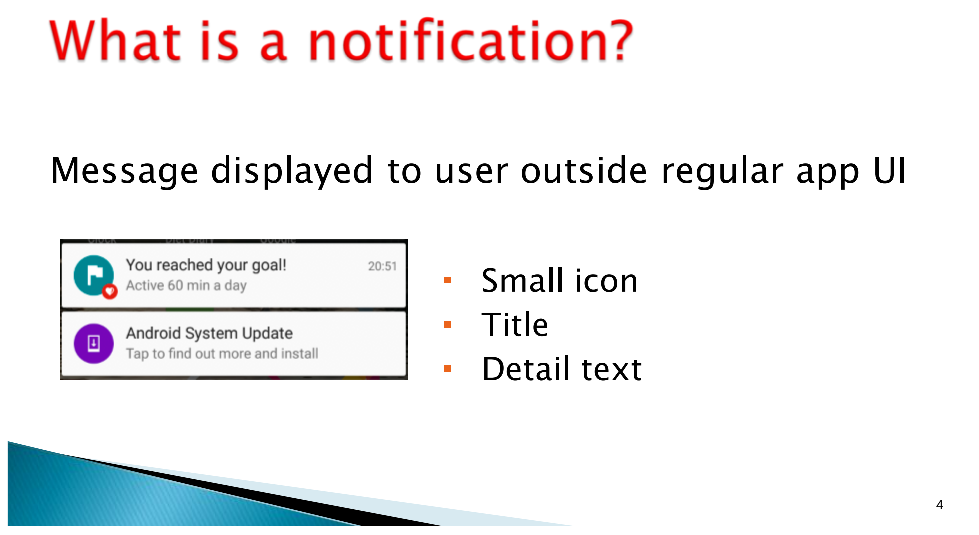
Stopping a service

- A **started service** must manage its own lifecycle
- If not stopped, will keep running and consuming resources
- The service must stop itself by calling stopSelf()
- Another component can stop it by calling stopService()
- **Bound service** is destroyed when all clients unbound
- **IntentService** is destroyed after onHandleIntent() returns

11

7.What is a notification? How can you create a notification? How can you clear a notification? (definition-2.5+creation(build+notify+register)-6.5-clear-1)

A **notification** is a message you can display to the user outside of your application's normal UI. When you tell the system to issue a notification, it first appears as an icon in the notification area. To see the details of the notification, the user opens the notification drawer. Both the notification area and the notification drawer are system-controlled areas that the user can view at any time.



What is a notification?

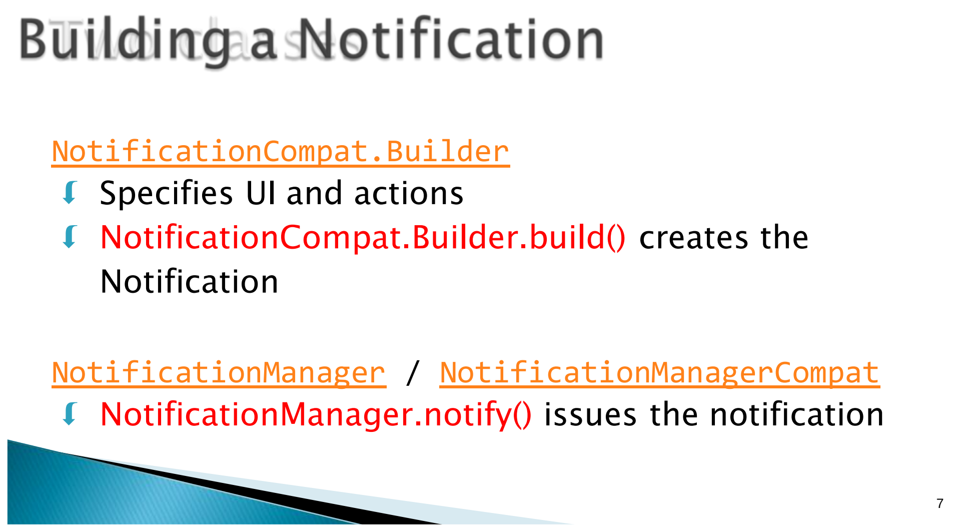
Message displayed to user outside regular app UI

You reached your goal! 20:51
Active 60 min a day

Android System Update
Tap to find out more and install

- Small icon
- Title
- Detail text

4



Building a Notification

NotificationCompat.Builder
- Specifies UI and actions
- NotificationCompat.Builder.build() creates the Notification

NotificationManager / NotificationManagerCompat
- NotificationManager.notify() issues the notification

7

Step 1 - Create Notification Builder

As a first step is to create a notification builder using *NotificationCompat.Builder.build()*. You will use Notification Builder to set various Notification properties like its small and large icons, title, priority etc.

- Define notification and set required attributes

```
NotificationCompat.Builder  mNotifyBuilder = new
NotificationCompat.Builder(this);
```

Step 2 - Setting Notification Properties

Once you have **Builder** object, you can set its Notification properties using Builder object as per your requirement. But this is mandatory to set at least following –

- A small icon, set by **setSmallIcon()**
- A title, set by **setContentTitle()**
- Detail text, set by **setContentText()**

You have plenty of optional properties which you can set for your notification. To learn more about them, see the reference documentation for NotificationCompat.Builder.

```
mNotifyBuilder.setContentTitle("You've been notified!");
mNotifyBuilder.setContentText("This is your notification
text.");
mNotifyBuilder.setSmallIcon(R.drawable.ic_android_black_24dp
);

Notification myNotification =mNotifyBuilder.build();`
```

Step 3 - Attach Actions

This is an optional part and required if you want to attach an action with the notification. An action allows users to go directly from the notification to an **Activity** in your application, where they can look at one or more events or do further work.

The action is defined by a **PendingIntent** containing an **Intent** that starts an Activity in your application. To associate the PendingIntent with a gesture, call the appropriate method of *NotificationCompat.Builder*. For example, if you want to start Activity when the user clicks the notification text in the notification drawer, you add the PendingIntent by calling **setContentIntent()**.

- A PendingIntent object helps you to perform an action on your applications behalf, often at a later time, without caring of whether or not your application is running. A PendingIntent is a description of an intent and target action to perform with it .

Step 1: Create intent

```
Intent notificationIntent =
    new Intent(this, MainActivity.class);
```

Step 2: Create PendingIntent

```
PendingIntent notificationPendingIntent =
    PendingIntent.getActivity(
              this,
              NOTIFICATION_ID,
              notificationIntent,
PendingIntent.FLAG_UPDATE_CURRENT);
```

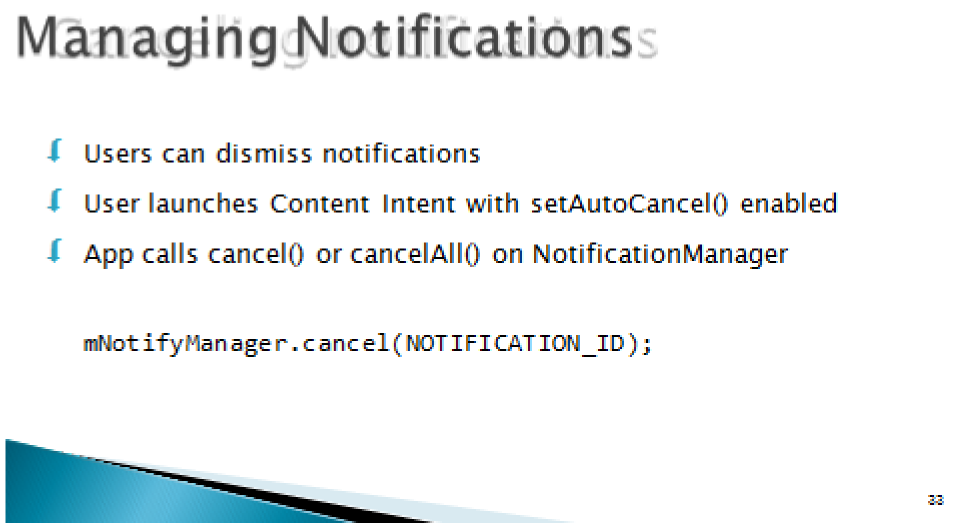
Step 3: Add to notification builder

```
.setContentIntent(notificationPendingIntent);
```

Step 4 - Issue the notification

Finally, you pass the Notification object to the system by calling NotificationManager.notify() to send your notification. Make sure you call **NotificationCompat.Builder.build()** method on builder object before notifying it. This method combines all of the options that have been set and return a new **Notification** object.

```
      NotificationManager mNotifyManager = (NotificationManager)
getSystemService(NOTIFICATION_SERVICE);

      mNotifyManager.notify(NOTIFICATION_ID,  myNotification);
```



Managing Notifications

- Users can dismiss notifications
- User launches Content Intent with setAutoCancel() enabled
- App calls cancel() or cancelAll() on NotificationManager

```
mNotifyManager.cancel(NOTIFICATION_ID);
```

33