Andhra Pradesh State Skill Development Corporation

Skill AP
APSSDC

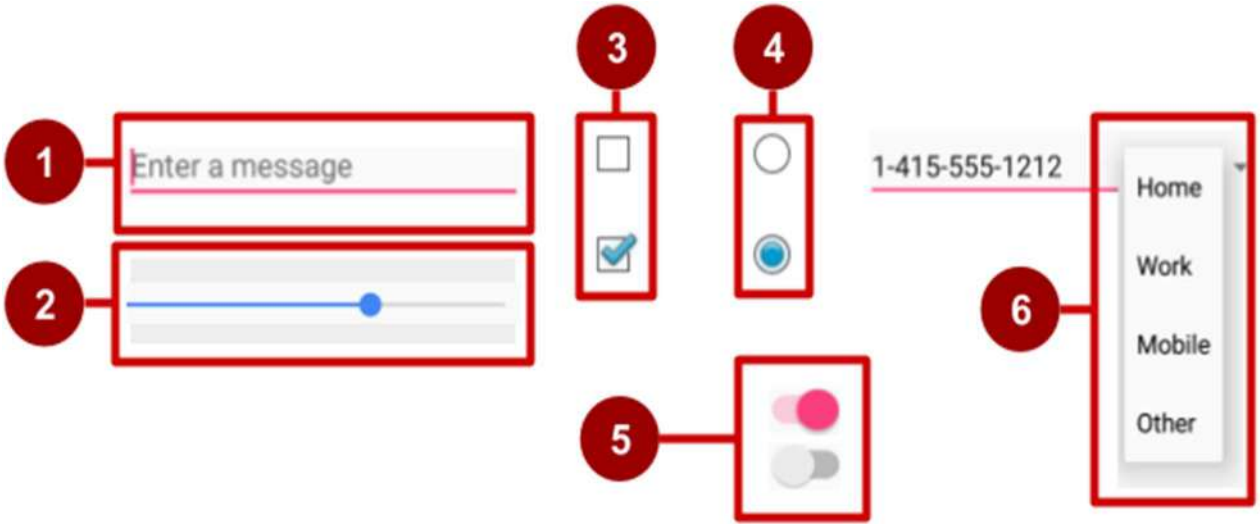
ANDROID APPLICATION DEVELOPMENT

INPUT CONTROLS

Input controls

This chapter introduces the Android *input controls*. Input controls are interactive elements in your app's UI that accept data input. Users input data to apps by entering text or numbers into fields using the on-screen keyboard. Users also select options from checkboxes, radio buttons, and drop-down menus, and they change settings and turn on or turn off certain features.

Android provides a variety of input controls for your UI. The figure below shows some popular ones.



In the figure above:

1. EditText field (subclass of TextView) for entering text using a keyboard
2. SeekBar for sliding left or right to a setting
3. CheckBox elements for selecting one or more options
4. RadioGroup of RadioButton elements for selecting one option
5. Switch for turning on or turning off an option
6. Spinner drop-down menu for selecting one option

When your app needs to get data from the user, try to make the process as easy for the user as it can be. For example, anticipate the source of the data, minimize the number of user gestures such as taps and swipes, and pre-fill forms when possible.

The user expects input controls to work in your app the same way they work in other apps. For example, users expect a *Spinner* to show a drop-down menu, and they expect text-editing fields to show a keyboard when tapped. Don't violate established expectations, or you'll make it harder for your users to use your app.

Input controls for making choices

Android offers ready-made input controls for the user to select one or more choices:

- CheckBox: Select one or more choices from a set of choices by tapping or clicking checkboxes.

- RadioGroup of radio buttons: Select one choice from a set of choices by clicking one circular "radio" button. Radio buttons are useful if you are providing only two or three choices.
- ToggleButton and Switch: Turn an option on or off.
- Spinner: Select one choice from a set of choices in a drop-down menu. A Spinner is useful for three or more choices, and takes up little room in your layout.

Input controls and the View focus

If your app has several UI input elements, which element gets input from the user first? * For example, if you have several *EditText* elements for the user to enter text, which element (that is, which View) receives the text? The View that "has the focus" receives user input.

Focus indicates which *View* is selected. The user can initiate focus by tapping on a *View*, for example a specific *EditText* element. You can define a focus order that defines how focus moves from one element to another when the user taps the Return key, Tab key, or arrow keys. You can also control focus programmatically by calling *requestFocus()* on any *View* that is focusable.

In addition to being focusable, input controls can be *clickable*. If a view's clickable attribute is set to *true*, then the view can react to click events. You can also make an element clickable programmatically.

What's the difference between focusable and clickable?

- A *focusable* view is allowed to gain focus from a touchscreen, external keyboard, or other input device.
- A *clickable* view is any view that reacts to being tapped or clicked.

Android-powered devices use many input methods, including directional pads (D-pads), trackballs, touchscreens, external keyboards, and more. Some devices, like tablets and smartphones, are navigated primarily by touch. Other device have no touchscreen. Because a user might navigate through your UI with an input device such as D-pad or a trackball, make sure you do the following:

- Make it visually clear which View has focus, so that the user knows where the input goes.
- Explicitly set the focus in your code to provide a path for users to navigate through the input elements using directional keys or a trackball.

Fortunately, in most cases you don't need to control focus yourself. Android provides "touch mode" for devices that respond to touch, such as smartphones and tablets. When the user begins interacting with the UI by touching it, only View elements with *isFocusableInTouchMode()* set to *true*, such as text input fields, are focusable. Other *View* elements that are touchable, such as *Button* elements, don't take focus when touched. If the user clicks a directional key or scrolls with a trackball, the device exits "touch mode" and finds a view to take focus.

Focus movement is based on a natural algorithm that finds the nearest neighbor in a given direction:

- When the user taps the screen, the topmost View under the tap is in focus, providing touch access for the child View elements of the topmost View.
- If you set an EditText view to a single line (such as the textPersonName value for the android:inputType attribute), the user can tap the right-arrow key on the on-screen keyboard to close the keyboard and shift focus to the next input control View based on what the Android system finds.

  The system usually finds the nearest input control in the same direction the user was navigating (up, down, left, or right).

  If there are multiple input controls that are nearby and in the same direction, the system scans from left to right, top to bottom.
- Focus can also shift to a different View if the user interacts with a directional control, such as a D-pad or trackball.

  You can influence the way Android handles focus by arranging input controls such as *EditText* elements in a certain layout from left to right and top to bottom, so that focus shifts from one to the other in the sequence you want.
  If the algorithm does not give you what you want, you can override it by adding the *nextFocusDown, nextFocusLeft, nextFocusRight,* and *nextFocusUp* XML attributes to your layout file:

1. Add one of these attributes to a View to decide where to go upon leaving the View—in other words, which View should be the next View.
2. Define the value of the attribute to be the id of the next View. For example:

```
<LinearLayout
  <!-- Other attributes... -->
  android:orientation="vertical" >

  <Button android:id="@+id/top"
      <!-- Other Button attributes... -->
      android:nextFocusUp="@+id/bottom" />

  <Button android:id="@+id/bottom"
      <!-- Other Button attributes... -->
      android:nextFocusDown="@+id/top"/>

</LinearLayout>
```

In a vertical *LinearLayout*, navigating up from the first *Button* would not ordinarily go anywhere, nor would navigating down from the second *Button*. But in the example above, the *top* Button has specified the bottom Button as the nextFocusUp (and vice versa), so the navigation focus will cycle from top-to-bottom and bottom-to-top.

To declare a View as focusable in your UI (when it is traditionally not), add the *android:focusable* XML attribute to the View in the layout, and set its value to true. You can also declare a View as focusable while in "touch mode" by setting android:focusableInTouchMode set to true.

You can also explicitly set the focus or find out which View has focus by using the following

methods:

- Call onFocusChanged to determine where focus came from.
- To find out which View currently has the focus, call Activity.getCurrentFocus(), or use ViewGroup.getFocusedChild() to return the focused child of a *View* (if any).
- To find the *View* in the hierarchy that currently has focus, use findFocus().
- Use requestFocus to give focus to a specific View.
- To change whether a View can take focus, call setFocusable.
- To set a listener that is notified when the View gains or loses focus, use setOnFocusChangeListener.

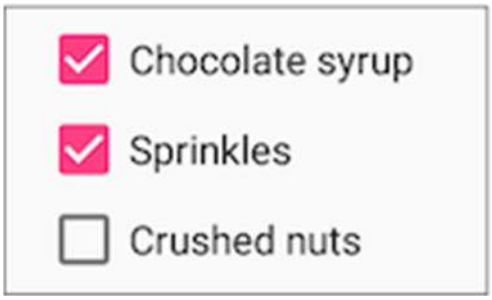
you learn more about focus with EditText elements.

Checkboxes

Use a set of checkboxes when you want the user to select any number of choices, including zero choices:

- Each checkbox is independent of the other boxes in the set, so selecting one box doesn't clear the other boxes. (If you want to limit the user's selection to one choice, use radio buttons.)
- A user can clear a checkbox that was already selected.

Users expect checkboxes to appear in a vertical list, like a to-do list, or side-by-side if the labels are short.



Each checkbox is a separate CheckBox element in your XML layout. To create multiple checkboxes in a vertical orientation, use a vertical LinearLayout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <CheckBox android:id="@+id/checkbox1_chocolate"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="ChocolateSyrup" />
    <CheckBox android:id="@+id/checkbox2_sprinkles"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Sprinkles" />
```

```
    <CheckBox android:id="@+id/checkbox3_nuts"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="CrushedNuts" />

</LinearLayout>
```

Typically programs retrieve the state of each *CheckBox* when a user taps or clicks a **Submit** or **Done** *Button* in the same *Activity*, which uses the *android:onClick* attribute to call a method such as *onSubmit()*:

```
<Button
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/submit"
  android:onClick="onSubmit"/>
```

The callback method—*onSubmit()* in the example above—must be *public*, return *void*, and define a *View* as a parameter (the view that was clicked). In this callback method you can determine whether a *CheckBox* is selected by using the isChecked() method (inherited from CompoundButton).

The *isChecked()* method returns *true* if there is a check mark in the box. For example, the following statement assigns *true* or *false* to *checked*, depending on whether the checkbox is checked:

```
public void onSubmit(View view) {
    CheckBox ch_java = findViewById(R.id.java);
    CheckBox ch_android = findViewById(R.id.android);
    StringBuilder builder = new StringBuilder();
    if(ch_java.isChecked()){
        builder.append(ch_java.getText().toString()+",");
    }

    if(ch_android.isChecked()){
        builder.append(ch_android.getText().toString());
    }
// Code to display the result...
}
```

Tip: To respond quickly to a *CheckBox*—such as display a message (like an alert), or show a set of further options—you can use the *android:onClick* attribute in the XML layout for each *CheckBox* to declare the callback method for that *CheckBox*. The callback method must be defined within the *Activity* that hosts this layout.

For more information about checkboxes, see Checkboxes in the Android developer documentation.

Radio buttons

Use radio buttons when you have two or more options that are mutually exclusive. When the user selects one, the others are automatically deselected. (If you want to enable more than one

selection from the set, use checkboxes.)

Users expect radio buttons to appear as a vertical list, or side-by-side if the labels are short. Each radio button is an instance of the RadioButton class. Radio buttons are normally placed within a RadioGroup in a layout. When several RadioButton elements are inside a RadioGroup, selecting one RadioButton clears all the others.

Add RadioButton elements to your XML layout within a RadioGroup:

```
<RadioGroup
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    >
    <RadioButton
      android:id="@+id/sameday"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onClick="onRadioButtonClicked"
      android:text="One" />
    <RadioButton
      android:id="@+id/nextday"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onClick="onRadioButtonClicked"
      android:text="Two" />
    <RadioButton
      android:id="@+id/pickup"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onClick="onRadioButtonClicked"
      android:text="Three" />
</RadioGroup>
```

Use the *android:onClick* attribute for each *RadioButton* to declare the click handler, which must be defined within the *Activity* that hosts the layout. In the layout above, clicking any *RadioButton* calls the same *onRadioButtonClicked()* method in the *Activity*. You could also create separate click handlers in the *Activity* for each *RadioButton*.

The click handler method must be *public*, return *void*, and define a *View* as its only parameter (the view that was clicked). The following shows one click handler, *onRadioButtonClicked()*, for all the *RadioButton* elements in the *RadioGroup*. It uses a *switch* case block to check the resource *id* for the *RadioButton* element to determine which one was checked:

```
public void onRadioButtonClicked(View view) {
  // Check to see if a button has been clicked.
  boolean checked = ((RadioButton) view).isChecked();
  // Check which radio button was clicked.
  switch(view.getId()) {
    case R.id.sameday:
      if (checked)
        // Code for same day service ...
        break;
    case R.id.nextday:
      if (checked)
        // Code for next day delivery ...
        break;
    case R.id.pickup:
      if (checked)
        // Code for pick up ...
        break;
  }
}
```

Tip: To give users a chance to review their radio button selection before the app responds, you could implement a **Submit** or **Done** button as shown previously with checkboxes, and remove the *android:onClick* attributes from the radio buttons. Then add the *onRadioButtonClicked()* method to the android:onClick attribute for the **Submit** or **Done** button.

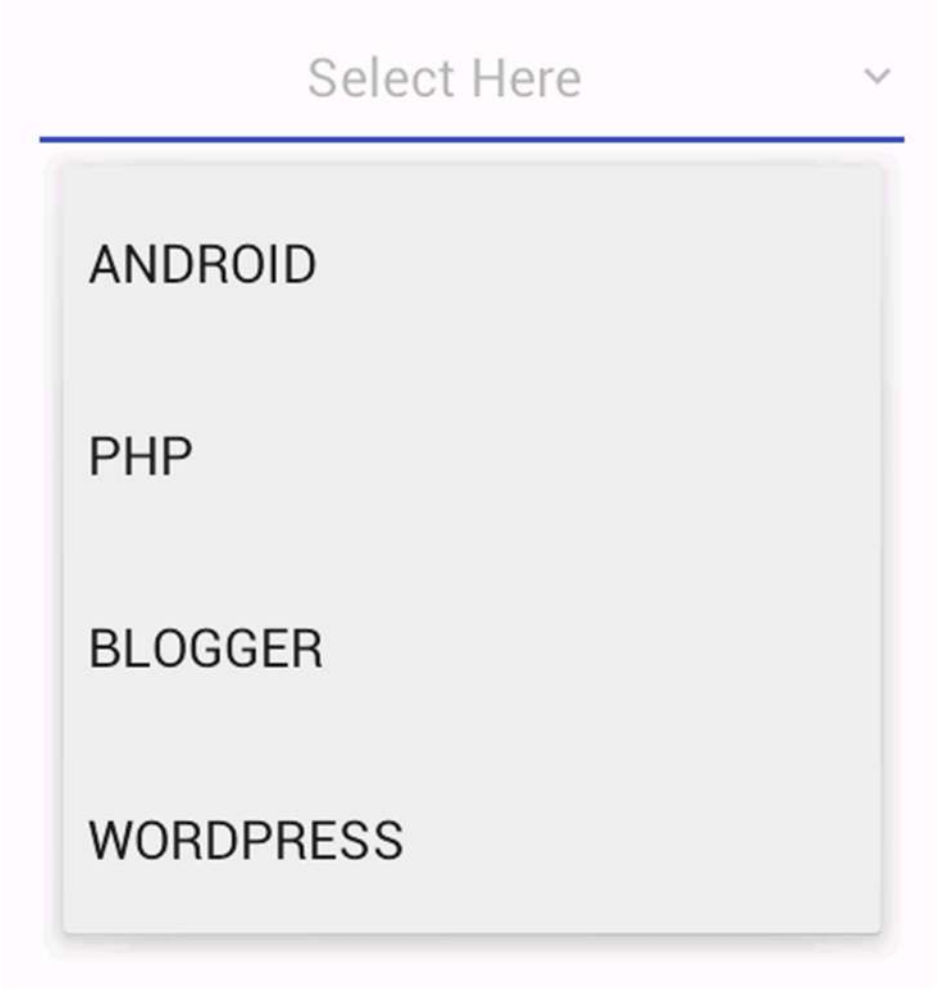
For more information about radio buttons, see Radio Buttons in the Android developer documentation.

Spinner

A Spinner provides a quick way for the user to select one value from a set. The user taps on the spinner to see a drop-down list with all available values.

A spinner works well when the user has more than three choices, because spinners scroll as needed, and a spinner doesn't take up much space in your layout. If you are providing only two or three choices and you have space in your layout, you might want to use radio buttons instead of a spinner.

Tip: For more information about spinners, see the Spinners guide.



If you have a long list of choices, a spinner might extend beyond your layout, forcing the user to scroll. A spinner scrolls automatically, with no extra code needed. However, making the user scroll through a long list (such as a list of countries) isn't recommended, because it can be hard for the user to select an item.

To create a spinner, use the Spinner class, which creates a View that displays individual spinner values as child View elements and lets the user pick one. Follow these steps:

1. Create a Spinner element in your XML layout, and specify its values using an array and an ArrayAdapter.
2. Create the *Spinner* and its adapter using the SpinnerAdapter class.
3. To define the selection callback for the Spinner, update the *Activity* that uses the *Spinner* to implement the AdapterView.OnItemSelectedListener interface.

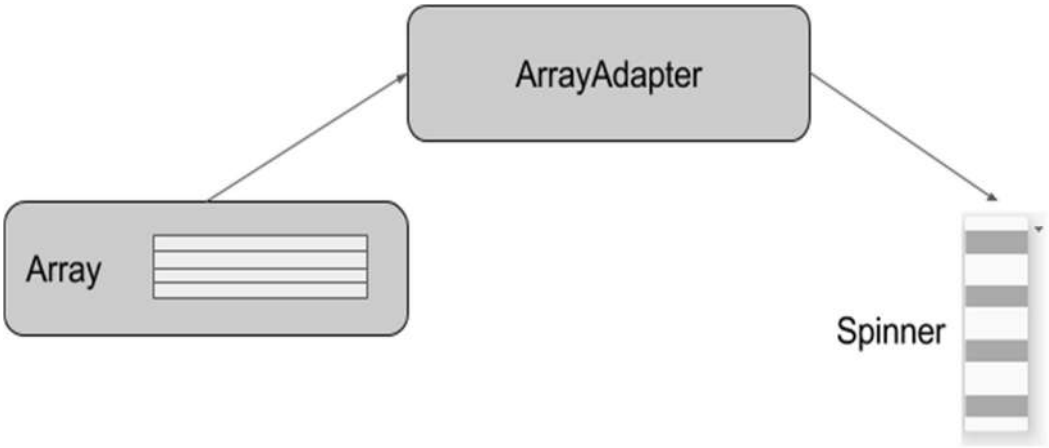
Create the Spinner UI element

To create a spinner in your XML layout, add a Spinner element, which provides the drop-down list:

```
<Spinner
  android:id="@+id/label_spinner"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"/>
```

Specify values for the Spinner

Add an adapter that fills the *Spinner* list with values. An adapter is like a bridge, or intermediary, between two incompatible interfaces. For example, a memory card reader acts as an adapter between the memory card and a laptop. You plug the memory card into the card reader, and plug the card reader into the laptop, so that the laptop can read the memory card.
The adapter takes the data set you've specified (an array in this example), and makes a *View* for each item in the data set (a *View* within the *Spinner*), as shown in the figure below.



The SpinnerAdapter class, which implements the *Adapter* class, allows you to define two different views: one that shows the data values in the *Spinner* itself, and one that shows the data in the drop-down list when the *Spinner* is touched or clicked.

The values you provide for the *Spinner* can come from any source, but must be provided through a *SpinnerAdapter*, such as an ArrayAdapter if the values are easily stored in an array. The following shows a simple array called *labels*array_ of predetermined values in the ***strings.xml*** file:

```
<string-array name="labels_array">
    <item>Home</item>
    <item>Work</item>
    <item>Mobile</item>
    <item>Other</item>
</string-array>
```

Tip: You can use a CursorAdapter if the values are provided from a source such as a stored file or a database. You learn more about stored data in another lesson.

Create the Spinner and its adapter

Create the *Spinner*, and set its listener to the *Activity* that implements the callback methods. The best place to do this is after the *Activity* layout is inflated in the *onCreate()* method. Follow these steps:

1. Instantiate a *Spinner* in the *onCreate()* method using the *label*spinner_ element in the layout, and set its listener *(spinner.setOnItemSelectedListener)* in the *onCreate()* method, as shown in the following code snippet:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  // ... Rest of onCreate code ...
  // Create the spinner.
  Spinner spinner = findViewById(R.id.label_spinner);
  if (spinner != null) {
        spinner.setOnItemSelectedListener(this);
  }
  // Create ArrayAdapter using the string array and default spinner layout.
```

The code snippet above uses findViewById() to find the Spinner by its id (label_spinner). It then sets the *onItemSelectedListener* to whichever Activity implements the callbacks (this) using the setOnItemSelectedListener() method

2. Continuing to edit the onCreate() method, add a statement that creates the ArrayAdapter with the string array (labels_array) using the Android-supplied Spinner layout for each item (layout.simple_spinner_item):

```
String[] branch = getResources().getStringArray(R.array.branches);
ArrayAdapter<String> arrayAdapter=new ArrayAdapter<>(this,android.R.layout.simple_list_item_1,branch);
```

3. Specify the layout for the Spinner choices to be simple_spinner_dropdown_item, and then apply the adapter to the Spinner:

```
spinner.setAdapter(arrayAdapter);
```

The snippet above uses setAdapter() to apply the adapter to the Spinner. You should use the simple_spinner_dropdown_item default layout, unless you want to define your own layout for the Spinner appearance.

Add code to respond to Spinner selections

When the user chooses an item from the spinner's drop-down list, here's what happens and how you retrieve the item:

1. The Spinner receives an on-item-selected event.
2. The event triggers the calling of the onItemSelected() callback method of the AdapterView.OnItemSelectedListener interface.
3. Retrieve the selected item in the Spinner using the getItemAtPosition() method of the *AdapterView* class:

```
spinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
      @Override
      public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
        //Implement Your Action
      }
      @Override
      public void onNothingSelected(AdapterView<?> parent) {
        //Implement Your Action
```

```
        }
});
```

The arguments for onItemSelected() are as follows:

parent AdapterView	The AdapterView where the selection happened
view View	The View within the AdapterView that was clicked
int pos	The position of the View in the adapter
long id	The row id of the item that is selected

Implement/override the *onNothingSelected()* callback method of the AdapterView.OnItemSelectedListener interface to do something if nothing is selected. For more information about using spinners, see Spinners in the Android developer documentation.

Practical Example:

activity_mail.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView 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:padding="10dp"
    tools:context=".UserInoutControls">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <ImageView
            android:layout_width="100dp"
            android:layout_height="100dp"
            android:layout_gravity="center"
            android:src="@mipmap/ic_launcher" />

        <EditText
            android:id="@+id/name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="10dp"
            android:hint="Enter your name"
            android:inputType="textCapWords" />

        <EditText
            android:id="@+id/email"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
```

```
    android:layout_marginTop="10dp"
    android:hint="Enter your emailid"
    android:inputType="textEmailAddress" />

<EditText
    android:id="@+id/mobile"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
    android:hint="Enter your mobileno"
    android:inputType="number" />

<EditText
    android:id="@+id/password"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
    android:hint="Enter your password"
    android:inputType="textPassword" />

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
    android:text="Gender"
    android:textSize="20sp" />

<RadioGroup
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
    android:orientation="horizontal">

    <RadioButton
        android:id="@+id/malerb"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Male" />

    <RadioButton
        android:id="@+id/femalerb"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Female" />
</RadioGroup>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dp"
```

```
        android:text="Technical Skills"
        android:textSize="20sp" />

    <CheckBox
        android:id="@+id/java"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:text="Java" />

    <CheckBox
        android:id="@+id/android"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:text="Android" />

    <Spinner
        android:id="@+id/branchsp"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:entries="@array/branches" />

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="displyData"
        android:text="Display" />

    <TextView
        android:id="@+id/result"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="ResultData"
        android:textSize="30sp" />

  </LinearLayout>
</ScrollView>
```

string.xml

```
<string-array name="branches">
    <item></item>
    <item>CSE</item>
    <item>ECE</item>
    <item>EEE</item>
    <item>IT</item>
    <item>Mech</item>
</string-array>
```

MainActivity.java

```
import android.os.Bundle;
import android.view.View;
import android.widget.CheckBox;
import android.widget.EditText;
import android.widget.RadioButton;
import android.widget.Spinner;
import android.widget.TextView;

import androidx.appcompat.app.AppCompatActivity;

public class UserInoutControls extends AppCompatActivity {

    EditText et_name, et_email, et_phone, et_pass;
    RadioButton r_male, r_female;
    CheckBox ch_java, ch_android;
    Spinner sp_branch;
    TextView tv_data;
    String gender;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        et_name = findViewById(R.id.name);
        et_email = findViewById(R.id.email);
        et_phone = findViewById(R.id.mobile);
        et_pass = findViewById(R.id.password);

        tv_data = findViewById(R.id.result);

        r_male = findViewById(R.id.malerb);
        r_female = findViewById(R.id.femalerb);

        ch_java = findViewById(R.id.java);
        ch_android = findViewById(R.id.android);

        sp_branch = findViewById(R.id.branchsp);

    }

    public void displyData(View view) {
        String name = et_name.getText().toString();
        String email = et_email.getText().toString();
        String phone = et_phone.getText().toString();
        String pass = et_pass.getText().toString();
        if (r_male.isChecked()) {
            gender = r_male.getText().toString();
        } else if (r_female.isChecked()) {
            gender = r_female.getText().toString();
```

```
        }
        StringBuilder builder = new StringBuilder();
        if (ch_java.isChecked()) {
            builder.append(ch_java.getText().toString() + ",");
        }

        if (ch_android.isChecked()) {
            builder.append(ch_android.getText().toString());
        }

        String branch = sp_branch.getSelectedItem().toString();

        tv_data.setText(name + "\n" + email + "\n" + phone + "\n" + pass + "\n" +
                gender + "\n" + builder.toString() + "\n" + branch);

    }
}
```

Output