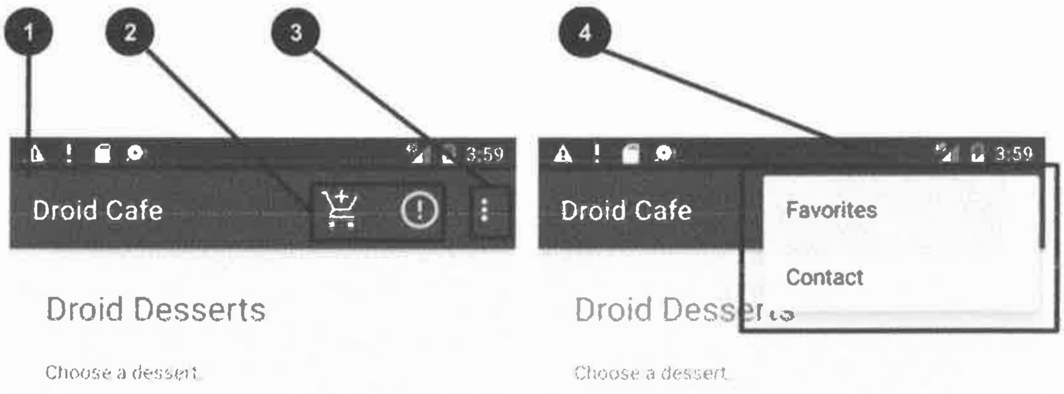
Practical 8: Menus and pickers

The ***app bar*** (also called the ***action bar***) is a **dedicated space at the top of each activity screen**. When you create an Activity from the **Basic Activity template**, Android Studio **includes an app bar**.

The ***options menu*** **in the app bar** usually provides **choices for navigation**, such as navigation to another activity in the app. The menu might also provide choices that affect the use of the app itself, for example ways to **change settings** or **profile information**, which usually **happens in a separate activity**.

In this practical you learn about setting up the app bar and options menu in your app, as shown in the figure below.



In the figure above:

1. **App bar**. The app bar includes the **app title**, the **options menu**, and the **overflow button**.
2. **Options menu action icons**. The first two options menu items appear as **icons** in the app bar.
3. **Overflow button**. The overflow button (**three vertical dots**) opens a menu that shows **more options menu items**.
4. **Options overflow menu**. After clicking the overflow button, **more options menu items** appear in the overflow menu.

Options menu items appear in the options overflow menu (see figure above). However, you can place some items as icons—as many as can fit—in the app bar. Using the app bar for the options menu makes your app consistent with other Android apps, allowing users to quickly understand how to operate your app and have a great experience.

Tip: To provide a familiar and consistent user experience, use the Menu APIs to present user actions and other options in your activities. See Menus for details.

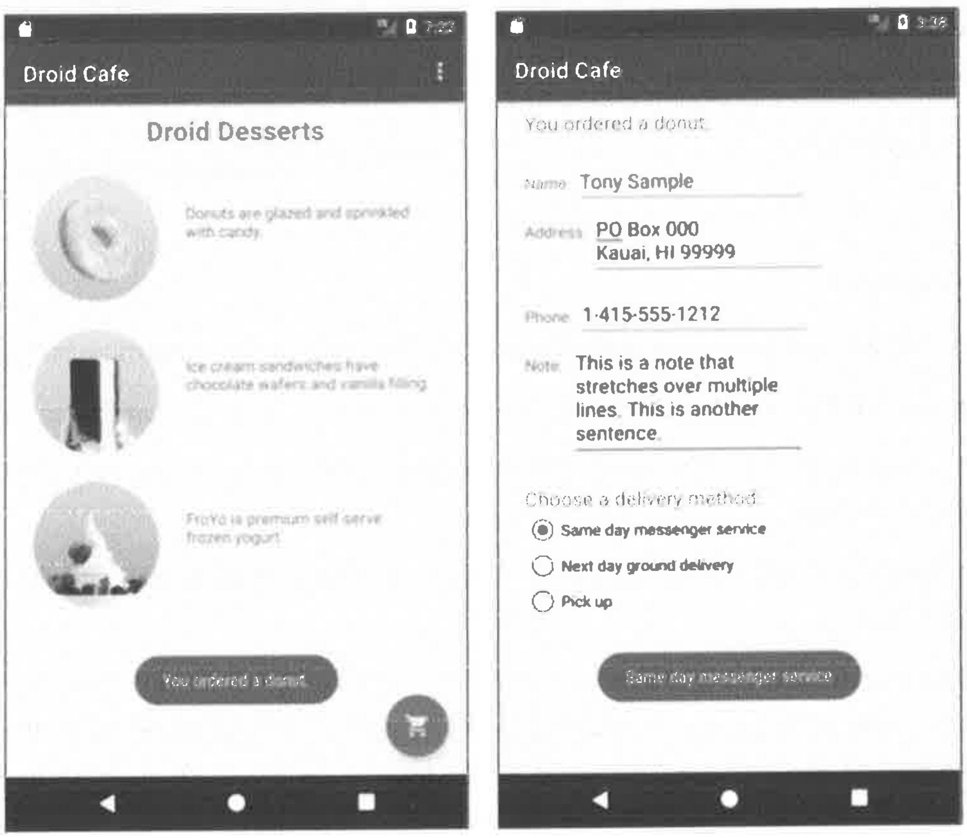
You also create an app that shows a **dialog to request a user's choice**, such as an alert that requires users to tap **OK** or **Cancel**. A ***dialog*** is a **window that appears on top of the display** or fills the display, interrupting the flow of activity.

Android provides **ready-to-use dialogs**, called ***pickers***, for **picking a time or a date**. You can use them to **ensure that your users pick a valid time or date that is formatted correctly and adjusted to the user's local time and date**. In this lesson you'll also create an app with the date picker.

- Continue adding features to the Droid Cafe project from the previous practical.
- Add menu items to the options menu.
- Add icons for menu items to appear in the app bar.
- Connect menu-item clicks to event handlers that process the click events.
- Use an alert dialog to request a user's choice.
- Use a date picker for date input.

App overview

In the previous practical you created an app called Droid Cafe, shown in the figure below, using the Basic Activity template. This template also provides a skeletal options menu in the app bar at the top of the screen.


Droid Cafe
Droid Desserts
Donuts are glazed and sprinkled with candy.
Ice cream sandwiches have chocolate wafers and vanilla filling
FroYo is premium self serve frozen yogurt
You ordered a donut.
Droid Cafe
Same day messenger service

You ordered a donut.

Name: Tony Sample

Address: PO Box 000
Kauai, HI 99999

Phone: 1-415-555-1212

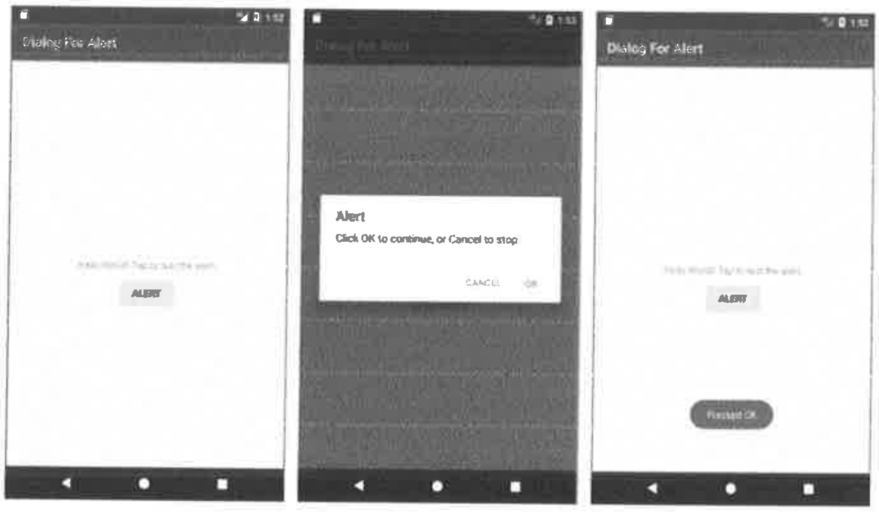
Note: This is a note that stretches over multiple lines. This is another sentence.

Choose a delivery method

[✓ Same day messenger service]
Next day ground delivery
Pick up

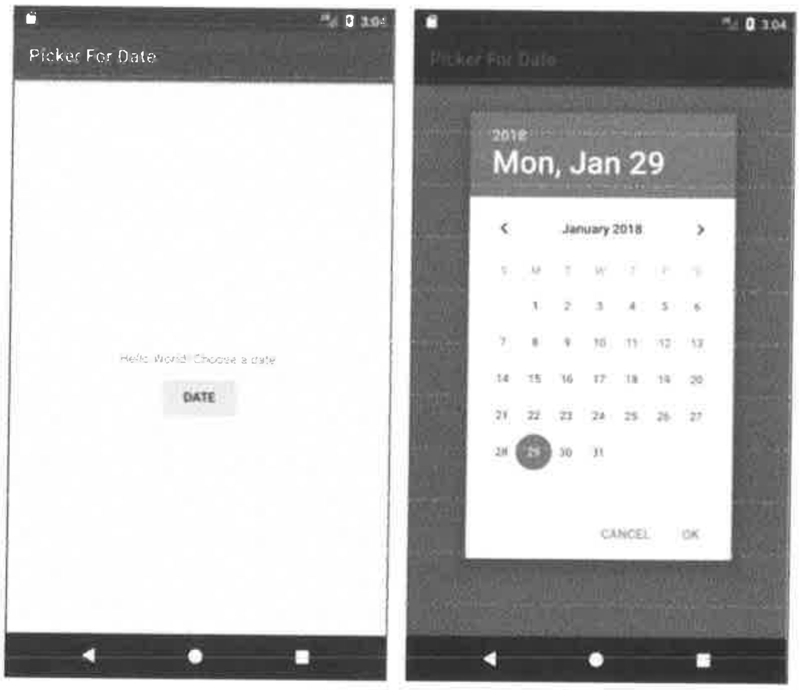
For this exercise you are using the **v7 appcompat** support library's Toolbar as an app bar, which works on the widest range of devices and also gives you room to customize your app bar later on as your app develops. To read more about design considerations for using the app bar, see Responsive layout grid in the Material Design specification.

You create a new app that displays an alert dialog. The dialog interrupts the user's workflow and requires the user to make a choice.



Click OK to continue, or Cancel to stop

You also create an app that provides a Button to show the date picker, and converts the chosen date to a string to show in a Toast message.



Task 1: Create the TwoActivities project

In this task you open the DroidCafeInput project from the previous practical and add menu items to the options menu in the app bar at the top of the screen.

1.1 Examine the code

Open the DroidCafeInput app from the practical on using input controls and examine the following layout files in the **res > layout** folder:

- **activity_main.xml**: The main layout for **MainActivity**, the first screen the user sees.
- **content_main.xml**: The layout for the content of the **MainActivity** screen, which (as you will see shortly) is *included* within **activity_main.xml**.
- **activity_order.xml**: The layout for **OrderActivity**, which you added in the practical on using input controls.

Follow these steps:

1. Open **content_main.xml** and click the **Text** tab to see the **XML code**. The **app:layout_behavior** for the **ConstraintLayout** is set to **@string/appbar_scrolling_view_behavior**, which controls how the **screen scrolls** in relation to the app bar at the top. (This string resource is defined in a generated file called **values.xml**, which you should not edit.)

   For more about scrolling behavior, see Android Design Support Library in the Android Developers Blog. For design practices involving scrolling menus, see Scrolling in the Material Design specification.

2. Open **activity_main.xml** and click the **Text** tab to see the **XML code** for the main layout, which uses a **CoordinatorLayout** layout with an embedded **AppBarLayout** layout.

   The **CoordinatorLayout** and the **AppBarLayout** tags require fully **qualified names** that specify **android.support.design**, which is the **Android Design Support Library**.

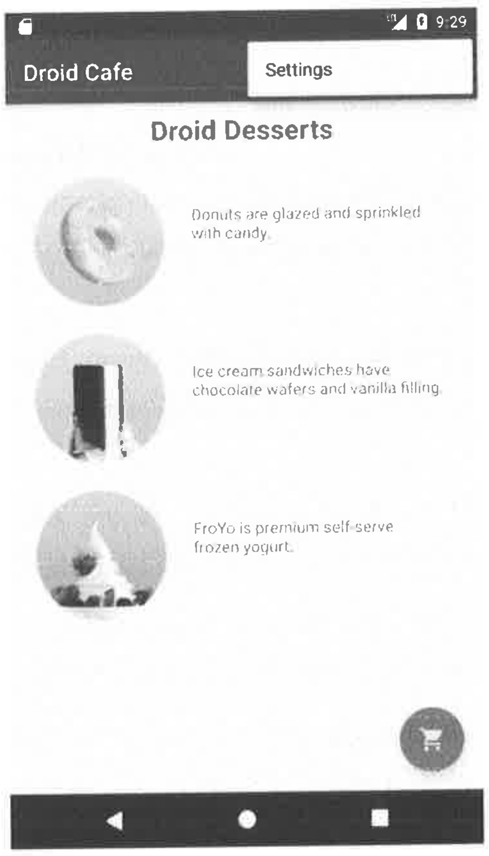
   AppBarLayout is like a vertical **LinearLayout.** It uses the **Toolbar class** in the **support library**, instead of the **native ActionBar**, to implement an app bar. The **Toolbar** within this layout has the **id toolbar**, and is also specified, like the **AppBarLayout**, with a **fully qualified name (android.support.v7.widget).**

The **app bar** is a section at the **top** of the display that can display the activity title, navigation, and other interactive items.

- The **native ActionBar behaves differently depending on the version** of Android running on the device. For this reason, if you are **adding an options menu**, you should **use the v7 appcompat support library's Toolbar** as an app bar.
- **Using the Toolbar** makes it **easy to set up an app bar** that works on the widest range of devices, and also gives you room to **customize your app bar later** on as your app develops. **Toolbar includes the most recent features**, and works for any device that can use the support library.

The **activity_main.xml** layout also uses an **include layout** statement to include the entire layout defined in **content_main.xml**. This separation of layout definitions makes it easier to change the layout's *content* apart from the layout's toolbar definition and coordinator layout. This is a best practice for separating your content (which may need to be translated) from the format of your layout.

3. Run the app. Notice the bar at the top of the screen showing the name of the app (Droid Cafe). It also shows the *action overflow* button (three vertical dots) on the right side. Tap the overflow button to see the options menu, which at this point has only one menu option, **Settings**.



Donuts are glazed and sprinkled with candy.

Ice cream sandwiches have chocolate wafers and vanilla filling.

FroYo is premium self-serve frozen yogurt.

4. Examine the **AndroidManifest.xml** file. The **.MainActivity activity** is set to use the **NoActionBar** theme. This theme is defined in the **styles.xml** file (open **app > res >values > styles.xml** to see it). Styles are covered in another lesson, but you can see that the **NoActionBar** theme sets the **windowActionBar** attribute to **false** (no window app bar), and the **windowNoTitle** attribute to **true** (no title). These values are set because you are defining the app bar with **AppBarLayout**, **rather than using an ActionBar**.

   - Using one of the **NoActionBar** themes **prevents the app from using the native ActionBar class** to provide the app bar.

5. Look at **MainActivity**, which **extends AppCompatActivity** and starts with the **onCreate() method**, which sets the content view to the **activity_main.xml** layout and sets **toolbar** to be the **Toolbar** defined in the layout. It then calls setSupportActionBar() and passes **toolbar** to it, setting the **toolbar** as the app bar for the **Activity**.

   For best practices about adding the app bar to your app, see Add the app bar.

1.2 Add more menu items to the options menu

You will add the following menu items to the options menu:

- **Order**: Navigate to **OrderActivity** to see the dessert order.
- **Status**: Check the status of an order.
- **Favorites**: Show favorite desserts.
- **Contact**: Contact the cafe. Because you don't need the existing **Settings** item, you will change **Settings** to **Contact**.

Android provides a standard XML format to define menu items. Instead of building a menu in your **Activity** code, you can **define a menu and all of its menu items in an XML menu resource**. You can then inflate the menu resource (load it as a Menu object) in your Activity:

1. Expand **res > menu** in the **Project > Android** pane, and open **menu_main.xml**. The only menu item provided from the template is **action_settings** (the **Settings** choice), which is defined as:

```
<item
  android:id="@+id/action_settings"
  android:orderInCategory="100"
  android:title="@string/action_settings"
  app:showAsAction="never" />
```

2. **Change** the following attributes of the **action_settings** item to make it the **action_contact** item (**don't change** the existing android:**orderInCategory** attribute):

Attribute	**Value**
android:id	**"@+id/action_contact"**
android:title	**"Contact"**
app:showAsAction	"never"

3. **Extract the hard-coded string** "Contact" into the string resource **action_contact**.
4. **Add** a new **menu item** using the <item> **tag** within the <menu> **block**, and give the item the following attributes:

Attribute	**Value**
android:id	**"@+id/action_order"**
android:orderInCategory	"10"
android:title	**"Order"**
app:showAsAction	"never"

The **android:orderInCategory** attribute specifies the order in which the menu items appear in the menu, with the **lowest number appearing higher** in the menu. The **Contact** item is set to **100**, which is a big number in order to specify that it **shows up at the bottom** rather than the top. You set the **Order** item to **10**, which **puts it above Contact**, and leaves **plenty of room in the menu for more items**.

5. **Extract the hard-coded string "Order"** into the string resource **action_order**.
6. **Add** two more **menu items** the same way with the following attributes:

Status item attribute	**Value**
android:id	**"@+id/action_status"**
android:orderInCategory	"20"
android:title	**"Status"**
app:showAsAction	"never"
Favorites item attribute	**Value**
android:id	**"@+id/action_favorites"**
android:orderInCategory	"30"

android:title	**"Favorites"**
app:showAsAction	"never"

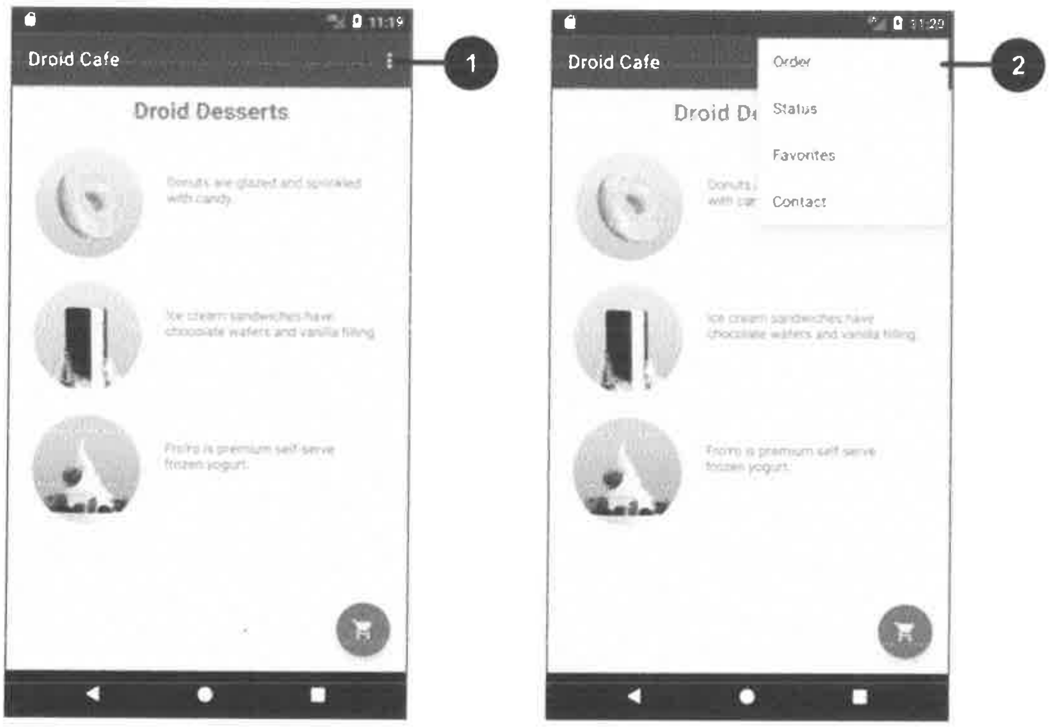
7. **Extract** "Status" into the **resource** action_status, and "Favorites" into the resource action_favorites.
8. You will **display a** Toast **message** with an action message **depending on which menu item the user selects**. Open **strings.xml** and **add the following string names and values** for these messages:

```
<string name="action_order_message">You selected Order.</string>
<string name="action_status_message">You selected Status.</string>
<string name="action_favorites_message">You selected Favorites.</string>
<string name="action_contact_message">You selected Contact.</string>
```

9. Open **MainActivity**, and **change** the **if statement** in the **onOptionsItemSelected() method** replacing the **id** action_settings with the new **id** action_order:

```
if (id == R.id.action_order)
```

Run the app, and tap the action overflow icon, shown on the left side of the figure below, to see the options menu, shown on the right side of the figure below. You will soon add callbacks to respond to items selected from this menu.



Droid Cafe
Droid Desserts
Donuts are glazed and sprinkled with candy.
Ice cream sandwiches have chocolate wafers and vanilla filling.
Froyo is premium self-serve frozen yogurt.
1
Droid Cafe
Order
Status
Favorites
Contact
2

In the figure above:

1. Tap the overflow icon in the app bar to see the options menu.
2. The options menu drops down from the app bar.

Notice the order of items in the options menu. You used the `android:orderInCategory` attribute to specify the priority of the menu items in the menu: The **Order** item is 10, followed by **Status** (20) and **Favorites** (30), and **Contact** is last (100). The following table shows the priority of items in the menu:

Menu item	orderInCategory attribute
Order	10
Status	20
Favorites	30
Contact	100

Task 2: Add icons for menu items

Whenever possible, you want to **show the most frequently used actions using icons in the app bar so the user can click them without having to first click the overflow icon**. In this task, you add icons for some of the menu items, and show some of menu items in the app bar at the top of the screen as icons.

In this example, (In your UI Design) assume that the **Order** and **Status** actions are the most frequently used. The **Favorites** action is occasionally used, and **Contact** is the least frequently used. You can set icons for these actions and specify the following:

- **Order** and **Status** should **always be shown** in the *app bar*.
- **Favorites should be shown** in the *app bar if it will fit; if not, it should appear in the overflow menu.*
- **Contact should not appear** in the app bar; it should only *appear in the overflow menu.*

2.1 Add icons for menu items

To specify icons for actions, you need to first **add the icons as image assets to the drawable folder** using the **same procedure** you used in the practical on using **clickable images**. You want to use the following icons (or similar ones):

- 🛒 **Order**: Use the same icon you added for the floating action button in the practical on using clickable images (ic_shopping_cart.png).
- ⓘ **Status**:
- ♥ **Favorites**:
- **Contact**: No need for an icon because it will appear only in the overflow menu.

For the **Status** and **Favorites** icons, follow these steps:

1. Expand **res** in the **Project > Android** pane, and right-click (or Control-click) the **drawable** folder.
2. Choose **New > Image Asset**. The Configure Image Asset dialog appears.
3. Choose **Action Bar and Tab Items** in the drop-down menu.
4. Change **ic_action_name** to another name (such as **ic_status_info** for the **Status** icon).
5. Click the clip art image (the Android logo next to **Clipart:**) to select a clip art image as the icon. A page of icons appears. Click the icon you want to use.

6. Choose **HOLO_DARK** from the **Theme** drop-down menu. This sets the icon to be white against a dark-colored (or black) background. Click **Next** and then click **Finish**.

   Tip: See Create app icons with Image Asset Studio for a complete description.

2.2 Show the menu items as icons in the app bar

To show menu items as icons in the app bar, use the **app:showAsAction attribute** in **menu_main.xml**. The following **values** for the attribute specify whether or not the action should appear in the app bar as an icon:

- "**always**": Always appears in the **app bar**. (If there isn't enough room it may overlap with other menu icons.)
- "**ifRoom**": Appears in the *app bar* ***if there is room***.
- "**never**": Never appears in the *app bar*; its text appears in the **overflow** menu.

Follow these steps to show some of the menu items as icons:

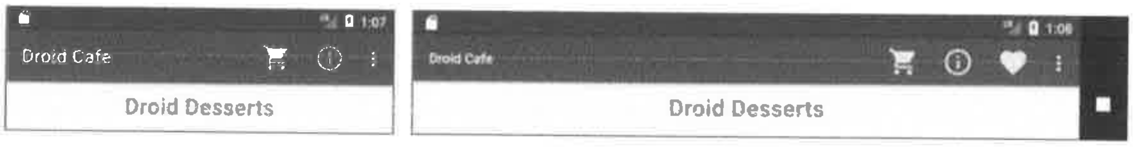
1. Open **menu_main.xml** again, and add the following attributes to the **Order**, **Status**, and **Favorites** items so that the first two (**Order** and **Status**) always appear, and the **Favorites** item appears only if there is room for it:

Order item attribute	**Old value**	**New value**
android:icon	*none*	**"@drawable/ic_shopping_cart"**
app:showAsAction	"never"	**"always"**

Status item attribute	**Old value**	**New value**
android:icon	*none*	**"@drawable/@drawable/ic_status_info"**
app:showAsAction	"never"	**"always"**

Favorites item attribute	**Old value**	**New value**
android:icon	*none*	**"@drawable/ic_favorite"**
app:showAsAction	"never"	**"ifRoom"**

2. Run the app. You should now see at least two icons in the app bar: the icon for **Order** and the icon for **Status** as shown on the left side of the figure below. (The **Favorites** and **Contact** options appear in the overflow menu.)
3. Rotate your device to the horizontal orientation, or if you're running in the emulator, click the **Rotate Left** or **Rotate Right** icons to rotate the display into the horizontal orientation. You should then see all three icons in the app bar for **Order**, **Status**, and **Favorites** as shown on the right side of the figure below.



How many action buttons will fit in the app bar? It depends on the orientation and the size of the device screen. Fewer buttons appear in a vertical orientation, as shown on the left side of the figure above, compared to a horizontal orientation as shown on the right side of the figure above. Action buttons may not occupy more than half of the main app bar width.

Task 3: Handle the selected menu item

In this task, you add a method to display a message about which menu item is tapped, and use the onOptionsItemSelected``() method to determine which menu item was tapped.

3.1 Create a method to display the menu choice

1. Open **MainActivity**.
2. If you haven't already added the following method (in another lesson) for displaying a **Toast message**, add it now. You will use it as the action to take for each menu choice. (Normally you would implement an action for each menu item such as starting another Activity, as shown later in this lesson.)

```
public void displayToast(String message) {
  Toast.makeText(getApplicationContext(), message,
                   Toast.LENGTH_SHORT).show();
}
```

3.2 Use the onOptionsItemSelected event handler

The **onOptionsItemSelected() method** handles selections from the options menu. You will add a **switch case block** to determine which menu item was selected and what action to take.

1. **Find** the **onOptionsItemSelected() method** provided by the template. The method **determines whether a certain menu item was clicked**, using the **menu item's id**. In the example below, the **id** is **action_order**:

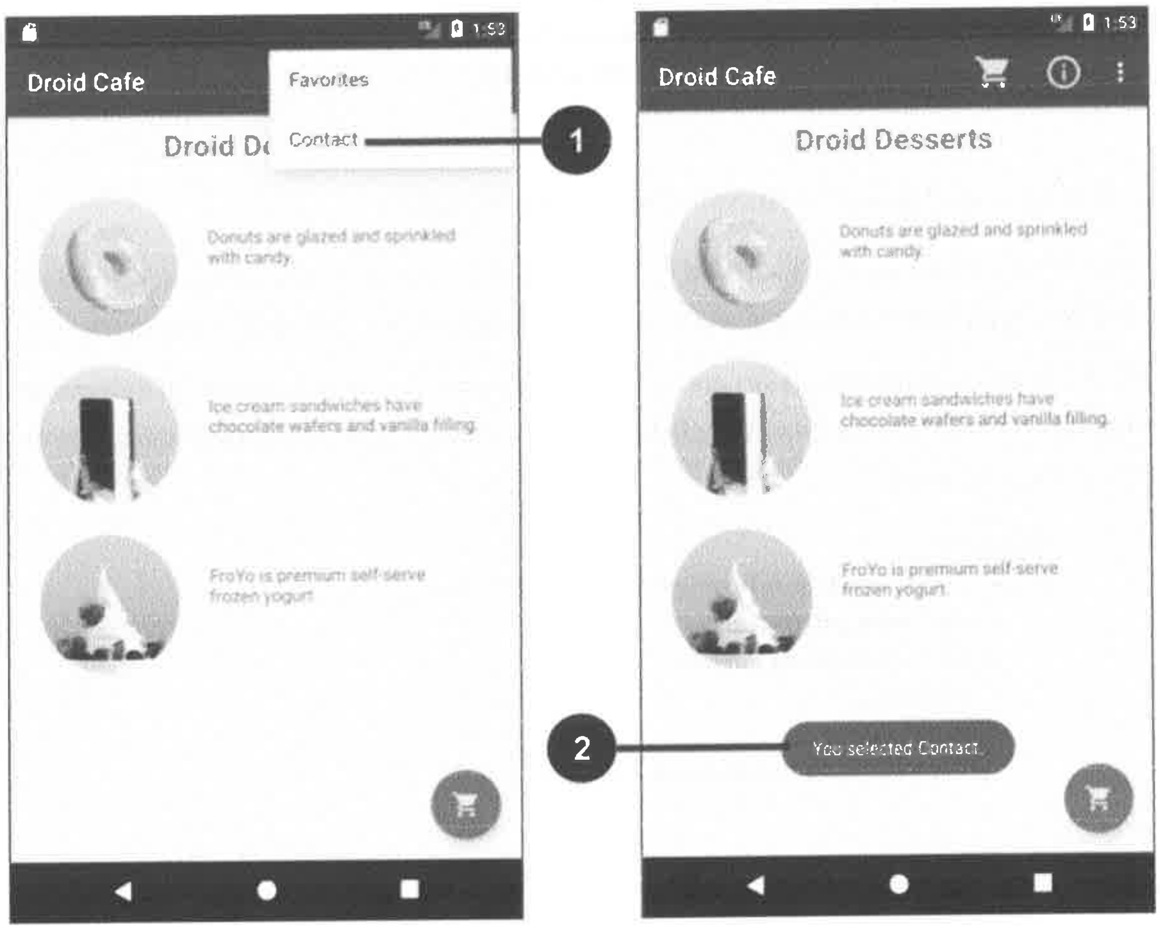
```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.action_order) {
       return true;
    }
    return super.onOptionsItemSelected(item);
}
```

2. **Replace** the **int id assignment statement** and the **if statement** with the following **switch case block**, which sets the appropriate **message** based on the menu item's **id**:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
       case R.id.action_order:
          displayToast(getString(R.string.action_order_message));
```

```
            return true;
        case R.id.action_status:
            displayToast(getString(R.string.action_status_message));
            return true;
        case R.id.action_favorites:
            displayToast(getString(R.string.action_favorites_message));
            return true;
        case R.id.action_contact:
            displayToast(getString(R.string.action_contact_message));
            return true;
        default:
            // Do nothing
    }
    return super.onOptionsItemSelected(item);
}
```

3. Run the app. You should now see a different Toast message on the screen, as shown on the right side of the figure below, based on which menu item you choose.



Droid Cafe
Favorites
Contact
Droid De
1
Donuts are glazed and sprinkled with candy.
Ice cream sandwiches have chocolate wafers and vanilla filling.
FroYo is premium self-serve frozen yogurt.
Droid Cafe
Droid Desserts
Donuts are glazed and sprinkled with candy.
Ice cream sandwiches have chocolate wafers and vanilla filling.
FroYo is premium self-serve frozen yogurt.
2
You selected Contact.

In the figure above:

1. Selecting the **Contact** item in the options menu.
2. The Toast message that appears.

3.3 Start an Activity from a menu item

Normally you would **implement an action for each menu item**, such as **starting another Activity**. Referring the snippet from the previous task, **change** the **code** for the **action_order** case to the following, **which start OrderActivity** (using the **same code you used for the floating action button** in the lesson on using **clickable images**):

```
switch (item.getItemId()) {
   case R.id.action_order:
      Intent intent = new Intent(MainActivity.this, OrderActivity.class);
      intent.putExtra(EXTRA_MESSAGE, mOrderMessage);
      startActivity(intent);
      return true;
   // ... code for other cases
}
```

Run the app. Clicking the shopping cart icon in the app bar (the **Order** item) takes you directly to the OrderActivity screen.

Task 3 solution code

Android Studio project: DroidCafeOptions

Task 4 Use a dialog to request a user's choice

You can provide a **dialog to request a user's choice**, such as an **alert** that requires users to tap **OK** or **Cancel**. A *dialog* is a window that appears on top of the display or fills the display, interrupting the flow of activity.

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**.

Tip: Use dialogs sparingly, because they interrupt the user's workflow. For best design practices, see the Dialogs Material Design guide. For code examples, see Dialogs in the Android developer documentation.

In this practical, you **use a Button to trigger a standard alert dialog.** In a real-world app, you might trigger an alert dialog based on **some condition**, or based on the **user tapping something**.

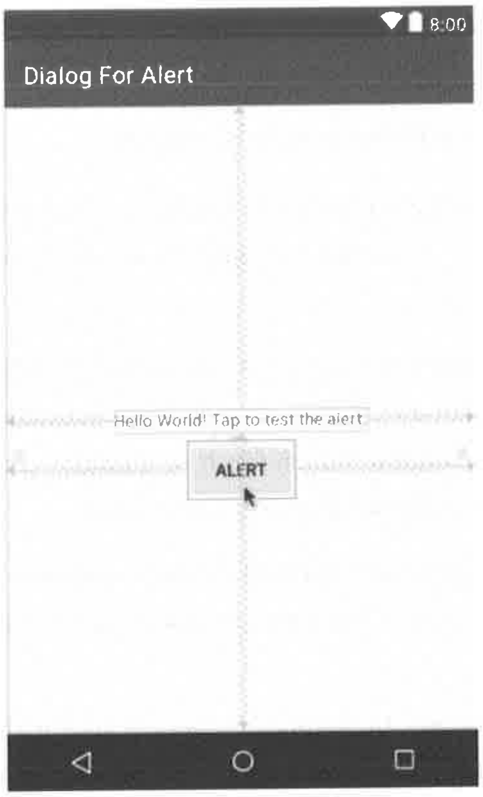
4.1 Create a new app to show an alert dialog

In this exercise, you build an **alert** with **OK** and **Cancel** buttons. The alert is triggered by the user **tapping a button**.

1. **Create** a **new project** called **Dialog For Alert** based on the **Empty Activity** template.
2. **Open** the **activity_main.xml** layout file to show the layout editor.
3. **Edit the TextView element** to say **Hello World! Tap to test the alert:** instead of "Hello World!"
4. **Add** a **Button** under the **TextView**. (Optional: Constrain the button to the bottom of the **TextView** and the sides of the layout, with margins set to 8dp.)
5. **Set** the text of the **Button** to **Alert**.
6. **Switch** to the **Text** tab, and **extract the text strings** for the **TextView** and **Button** to **string resources.**
7. Add the **android:onClick attribute** to the button **to call the click handler onClickShowAlert().** After you enter it, the click handler is underlined in **red** because it **has not yet been created**.

```
android:onClick="onClickShowAlert"
```

You now have a layout similar to the following:



4.2 Add an alert dialog to the main activity

The ***builder* design pattern** makes it easy to create an object from a class that has a lot of required and optional attributes and would therefore require a lot of parameters to build. Without this pattern, you would have to create constructors for combinations of required and optional attributes; with this pattern, the code is easier to read and maintain. For more information about the builder design pattern, see Builder pattern.

The **builder class** is usually a static member class of the class it builds. Use AlertDialog.Builder to build a standard alert dialog, with setTitle() to set its title, setMessage() to set its message, and setPositiveButton() and setNegativeButton() to set its buttons.

To make the alert, you need to **make an object** of **AlertDialog.Builder**. You will **add** the **onClickShowAlert() click handler** for the **Alert Button**, which makes this object as its first order of business. **That means that the dialog will be created only when the user clicks the Alert Button.** While this coding pattern is logical for using a Button to test an alert, for other apps you may want to create the dialog in the **onCreate() method** so that it is always available for other code to trigger it.

1. Open **MainActivity** and add the beginning of the **onClickShowAlert() method**:

```
public void onClickShowAlert(View view) {
   AlertDialog.Builder myAlertBuilder = new
                     AlertDialog.Builder(MainActivity.this);
   // Set the dialog title and message.
}
```

   If **AlertDialog.Builder** is **not recognized** as you enter it, click the **red light bulb icon**, and choose the support library version (**android.support.v7.app.AlertDialog**) for **importing into your Activity.**

2. **Add** the code to set the **title** and the message for the alert dialog to **onClickShowAlert()** after the comment:

```
// Set the dialog title and message.
myAlertBuilder.setTitle("Alert");

myAlertBuilder.setMessage("Click OK to continue, or Cancel to stop:");
// Add the dialog buttons.
```

3. **Extract the strings** above to string resources as **alert_title** and **alert_message**.

4. Add the **OK** and **Cancel** buttons to the alert with **setPositiveButton()** and **setNegativeButton()** methods:

```
// Add the dialog buttons.
myAlertBuilder.setPositiveButton("OK", new
                     DialogInterface.OnClickListener() {
   public void onClick(DialogInterface dialog, int which) {
      // User clicked OK button.
      Toast.makeText(getApplicationContext(), "Pressed OK",
                  Toast.LENGTH_SHORT).show();
   }
});

myAlertBuilder.setNegativeButton("Cancel", new
                     DialogInterface.OnClickListener() {
   public void onClick(DialogInterface dialog, int which) {
      // User cancelled the dialog.
      Toast.makeText(getApplicationContext(), "Pressed Cancel",
                  Toast.LENGTH_SHORT).show();
   }
});
// Create and show the AlertDialog.
```

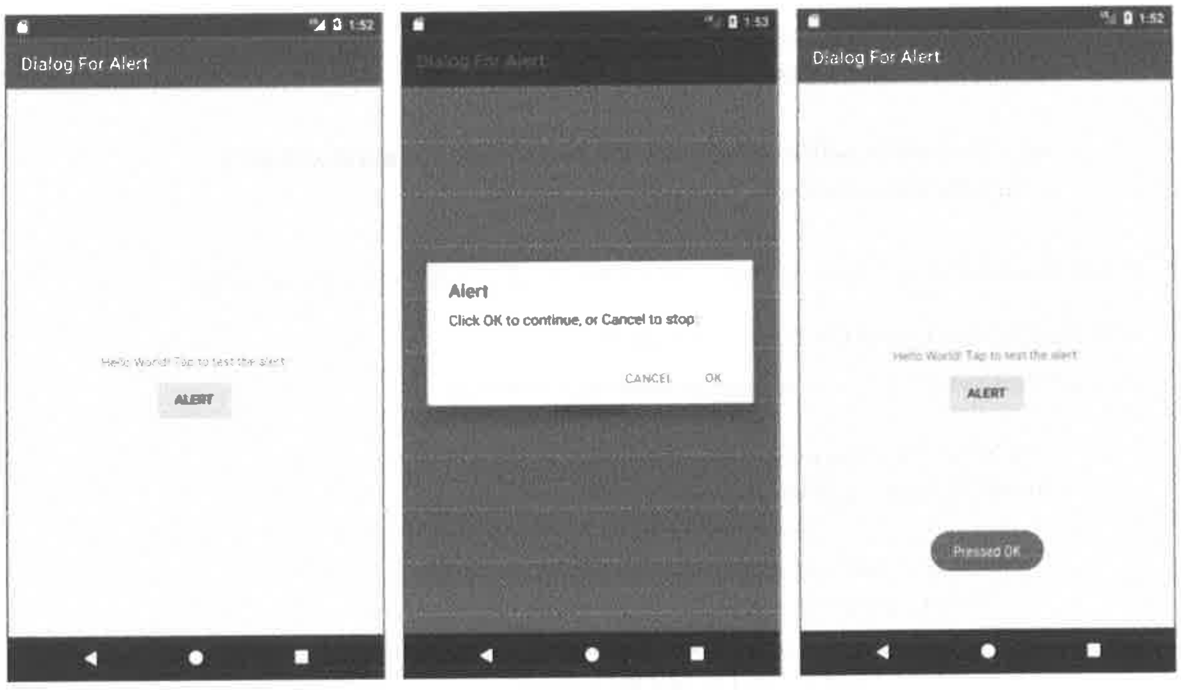
After the user taps the **OK** or **Cancel** button in the alert, you can grab the user's selection and use it in your code. In this example, you display a Toast message.

5. **Extract the strings** for **OK** and **Cancel** to string resources as **ok_button** and **cancel_button**, and **extract the strings** for the **Toast messages**.
6. At the end of the **onClickShowAlert() method**, **add show()**, which creates and then displays the alert dialog:

```
// Create and show the AlertDialog.
myAlertBuilder.show();
```

7. Run the app.

   You should be able to tap the **Alert** button, shown on the left side of the figure below, to see the alert dialog, shown in the center of the figure below. The dialog shows **OK** and **Cancel** buttons, and a Toast message appears showing which one you pressed, as shown on the right side of the figure below.


Dialog For Alert
Hello World! Tap to test the alert
ALERT
Alert
Click OK to continue, or Cancel to stop
CANCEL
OK
Dialog For Alert
Hello World! Tap to test the alert
ALERT
Pressed OK

Task 4 solution code

Android Studio project: DialogForAlert

Task 5: Use a picker for user input

Android provides **ready-to-use dialogs**, called ***pickers***, for **picking a time or a date**. You can use them to **ensure that your users pick a valid time or date that is formatted correctly and adjusted to the user's local time and date**. Each picker provides controls for selecting each part of the time (hour, minute, AM/PM) or date (month, day, year). You can read all about setting up pickers in Pickers.

In this task you'll **create a new project and add the date picker**. You will also learn how to use a Fragment, which is a **behavior** or a **portion of a UI within an Activity**. It's like **a mini-Activity within the main Activity**, with its own lifecycle, and it's **used for building a picker**. All the work is done for you. To learn about the Fragment class, see Fragments in the API Guide.

One benefit of using a **Fragment** for a picker is that you can **isolate the code sections** for **managing the date and the time for various locales that display date and time in different ways**.

The best practice to show a picker is to use an **instance of** DialogFragment, which is a **subclass of Fragment**. A **DialogFragment** displays a dialog window floating on top of the **Activity** window.

In this exercise, you'll add a Fragment for the picker dialog and use DialogFragment to manage the dialog lifecycle.

Tip: Another benefit of using a **`Fragment`** for a picker is that you can implement different layout configurations, such as a basic dialog on handset-sized displays or an embedded part of a layout on large displays.

5.1 Create a new app to show a date picker

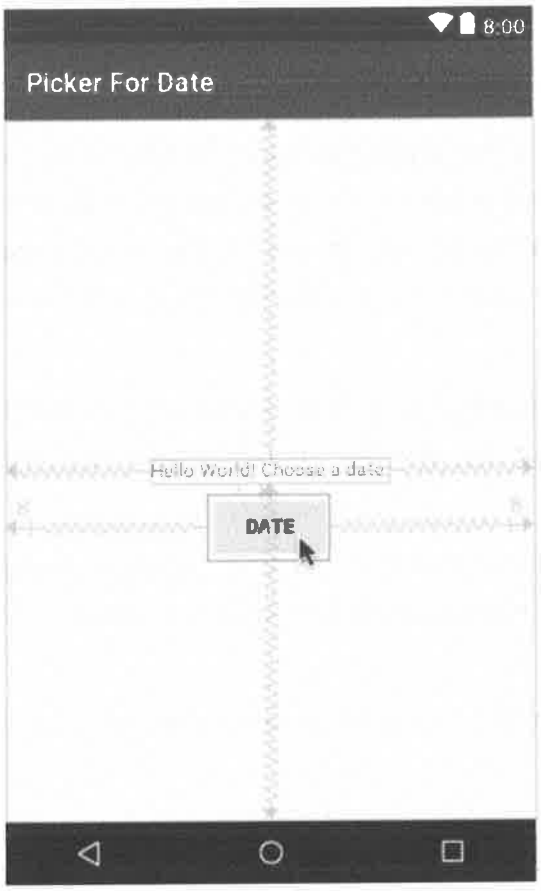
To start this task, **create an app** that provides a **Button** to **show the date picker**.

1. **Create** a **new project** called **Picker For Date** based on the **Empty Activity** template.
2. **Open** the **activity_main.xml** layout file to show the layout editor.
3. **Edit** the **TextView** element's "Hello World!" text to **Hello World! Choose a date:**.
4. **Add** a **Button** underneath the **TextView**. (Optional: Constrain the Button to the bottom of the **TextView** and the sides of the layout, with margins set to 8dp.)
5. **Set** the text of the **Button** to **Date**.

6. **Switch** to the **Text** tab, and **extract the strings** for the **TextView** and **Button** to **string resources**.
7. **Add** the **android:onClick** attribute to the **Button** to **call the click handler showDatePicker()**. After entering it, the click handler is underlined in **red** because it **has not yet been created**.

```
android:onClick="showDatePicker"
```

You should now have a layout similar to the following:



5.2 Create a new fragment for the date picker

In this step, you add a **Fragment** for the **date picker**.

1. Expand **app > java > com.example.android.pickerfordate** and select **MainActivity**.
2. Choose **File > New > Fragment > Fragment (Blank)**, and name the fragment **DatePickerFragment**. **Clear all three checkboxes** so that you **don't create a layout XML**, **include fragment factory methods**, or **include interface callbacks**. You **don't need to create a layout for a standard picker**. Click **Finish**.

3. **Open DatePickerFragment** and **edit** the **DatePickerFragment class** definition to extend **DialogFragment** and implement DatePickerDialog.OnDateSetListener to **create a standard date picker with a listener**. See Pickers for more information about extending DialogFragment for a date picker:

```
public class DatePickerFragment extends DialogFragment
                implements DatePickerDialog.OnDateSetListener {
```

   As you enter **DialogFragment** and **DatePickerDialog.OnDateSetListener**, Android Studio automatically adds several **import statements** to the import block at the top, including:

```
import android.app.DatePickerDialog;
import android.support.v4.app.DialogFragment;
```

   In addition, a **red bulb icon appears** in the left margin after a few seconds.

4. **Click the red bulb icon** and choose **Implement methods** from the popup menu. A dialog appears with onDateSet() already selected and the **Insert @Override** option selected. Click **OK** to create the empty **onDateSet() method**. This method will be called when the user sets the date.

   After adding the empty **onDateSet() method**, Android Studio automatically adds the following in the import block at the top:

```
import android.widget.DatePicker;
```

   The **onDateSet() parameters** should be **int i, int i1**, and **int i2**. Change the names of these parameters to ones that are **more readable**:

```
public void onDateSet(DatePicker datePicker,
                        int year, int month, int day)
```

5. **Remove** the empty **public DatePickerFragment()** public constructor.

6. **Replace** the entire **onCreateView() method** with onCreateDialog() that **returns Dialog**, and annotate **onCreateDialog()** with **@NonNull** to indicate that the **return value Dialog can't be null**. Android Studio displays a **red bulb** next to the method because **it doesn't return anything yet**.

```
@NonNull
@Override
public Dialog onCreateDialog(Bundle savedInstanceState) {
}
```

7. **Add** the following **code** to **onCreateDialog()** to **initialize** **the year, month, and day** from Calendar, and **return** the dialog and these values to the **Activity**. As you enter **Calendar.getInstance()**, specify the import to be **java.util.Calendar**.

```
// Use the current date as the default date in the picker.
final Calendar c = Calendar.getInstance();
int year = c.get(Calendar.YEAR);
int month = c.get(Calendar.MONTH);
int day = c.get(Calendar.DAY_OF_MONTH);

// Create a new instance of DatePickerDialog and return it.
return new DatePickerDialog(getActivity(), this, year, month, day);
```

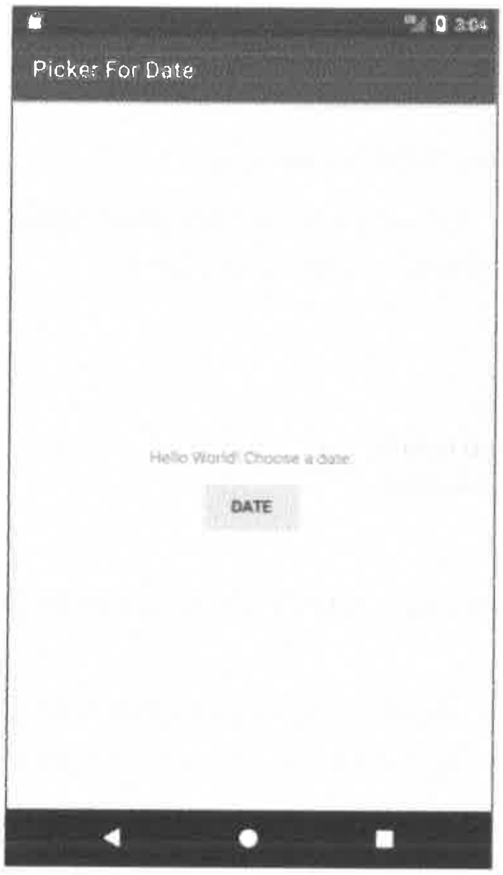
5.4 Modify the main activity

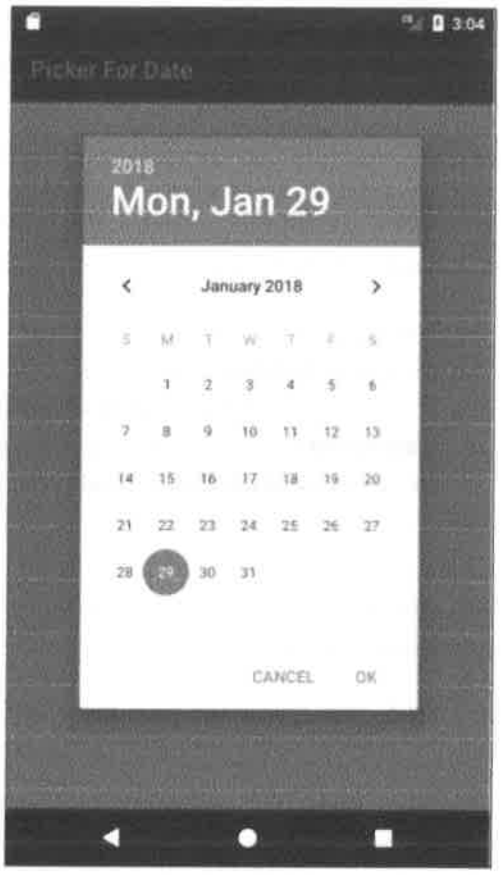
While much of the code in **MainActivity.java** stays the same, you need to **add** a method that **creates an instance of FragmentManager** to **manage the Fragment** and show the date picker.

1. **Open MainActivity**.
2. **Add** the **showDatePickerDialog()** handler for the **Date** Button. It creates an instance of **FragmentManager** using getSupportFragmentManager() to manage the Fragment automatically, and to show the picker. For more information about the Fragment class, see Fragments.

```
public void showDatePicker(View view) {
      DialogFragment newFragment = new DatePickerFragment();
      newFragment.show(getSupportFragmentManager(),"datePicker");
}
```

3. **Extract the string "datePicker"** to the **string resource datepicker**.
4. **Run** the app. You should see the date picker after tapping the **Date** button.





5.5 Use the chosen date

In this step you **pass the date back** to **MainActivity.java**, and **convert the date to a string** that you can **show in a Toast message**.

1. Open **MainActivity** and **add** an empty **processDatePickerResult() method** that takes the **year, month,** and **day** as arguments:

```
public void processDatePickerResult(int year, int month, int day) {
}
```

2. **Add** the following **code** to the **processDatePickerResult() method** to **convert the month, day, and year to separate strings**, and to **concatenate the three strings with slash marks** for the U.S. date format:

```
String month_string = Integer.toString(month+1);   // month start with 1
String day_string = Integer.toString(day);
String year_string = Integer.toString(year);
String dateMessage = (month_string +
                "/" + day_string + "/" + year_string);
```

The **month** integer returned by the **date picker starts counting at 0 for January**, so you **need to add 1** to it to show **months starting at 1**.

3. **Add** the following after the code above to display a Toast message:

```
Toast.makeText(this, "Date: " + dateMessage,
                    Toast.LENGTH_SHORT).show();
```

4. **Extract the hard-coded string "Date: "** into a **string resource named date**.

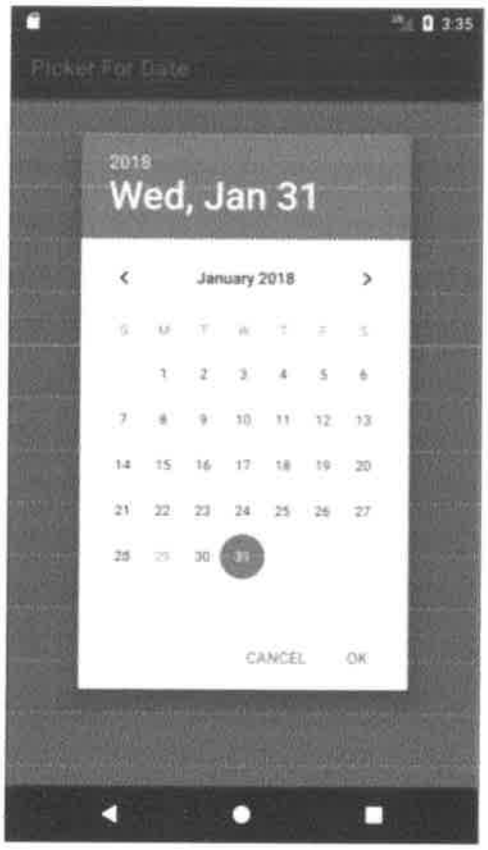
5. Open **DatePickerFragment**, and **add** the following to the **onDateSet() method** to **invoke** processDatePickerResult() in **MainActivity** and **pass it** the **year, month,** and **day**:

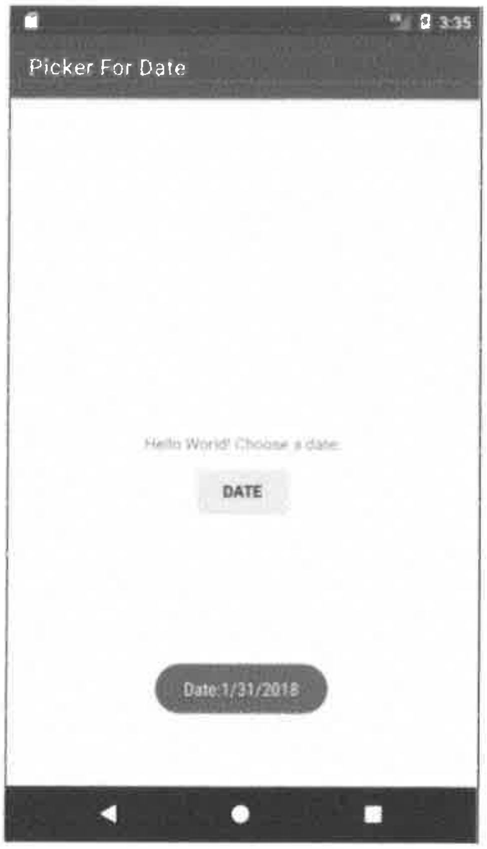
```
@Override
public void onDateSet(DatePicker datePicker,
                      int year, int month, int day) {
   MainActivity activity = (MainActivity) getActivity();
   activity.processDatePickerResult(year, month, day);
}
```

You use **getActivity()** which, when used in a **Fragment**, **returns the Activity the Fragment is currently associated with.**

- You need this because you can't call a method in MainActivity without the context of MainActivity (you would have to use an intent instead, as you learned in another lesson).
- The Activity inherits the context, so you can use it as the context for calling the method (as in activity.processDatePickerResult).

6. Run the app. After selecting the date, the date appears in a Toast message as shown on the right side of the following figure.





Hello World! Choose a date

Date:1/31/2018

Task 5 solution code

Android Studio project: PickerForDate