Menus
Three options (I)

- Android apps should provide an app bar to present common user actions

- Options menu and app bar
  - Is the primary collection of menu items (actions that have a global impact) for an activity
  - On Android 3.0 and higher, these items are presented by the app bar as a combination of on-screen action items and overflow options
Three options (II)

• **Context menu and contextual action mode**
  - It is a floating menu that appears when the user performs a long-click on an element and provides actions that affect the selected content or context frame
  - On Android 3.0 and higher, you should instead use the contextual action mode to display action items in a bar at the top of the screen and allow the user to select multiple items

• **Popup menu**
  - It displays a list of items in a vertical list that is anchored to the view that invoked the menu
    - Good for providing an overflow of actions that relate to specific content
Menus

• Must be properly designed
  – If they become too big, they do not fit
  – Should guide the user properly
  – Should exploit the current state/context

• Two ways
  – XML: file in res/menu and inflate it (reuse)
  – Java: directly part of the activity’s code
Usual XML definition

- Each menu entry is specified by an ID, a label, and an icon (optional)
  - Method onCreateOptionsMenu() inflates the menu
    - Specific actions must then be implemented
  - One can add a submenu to an item in any menu (except a submenu)
  - A menu group is a collection of menu items that share properties

```xml
<menu xmlns:android="...">
  <item android:id="@+id/newgame"
       android:icon="@drawable/ic_launcher"
       android:title="New Game"
       android:showAsAction="ifRoom"/>
  <item android:id="@+id/help"
       android:icon="@drawable/ic_launcher"
       android:title="Help" />
</menu>
```
Within the activity

```java
@override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.main, menu);
    return true;
}

@override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle item selection
    switch (item.getItemId()) {
    case R.id.newgame:
        newGame();
        return true;
    case R.id.help:
        showHelp();
        return true;
    default:
        return super.onOptionsItemSelected(item);
    }
}
```
Modifications

• To modify the options menu, use method onPrepareOptionsMenu()
  — This method passes the Menu object as it currently exists so we can modify it, such as add, remove, or disable items

• Starting from Android 3.0, when we want to perform a menu update, we must call invalidateOptionsMenu() to ask the system to call onPrepareOptionsMenu()
Contextual menus

• Most often used for items in a ListView, GridView, or other view collections in which the user can perform direct actions on each item.
Floating Context Menu

• Register the View by calling registerForContextMenu() and pass it the View
  – If you pass an entire ListView or GridView, you register all of its items

• Implement the onCreateContextMenu() method in your Activity to create/inflate the menu

• Implement onContextItemSelected() to perform the appropriate actions
Contextual Action Mode

• On individual views
  – Implement the ActionMode.Callback interface to specify the actions for the contextual action bar
  – Call startActionMode() when appropriate, such as in response to a long-click on a View

• On groups of items in a ListView or GridView
  – To perform an action on all of them
  – Implement the AbsListView.MultiChoiceModeListener interface and set it for the view group with setMultiChoiceModeListener() 
  – Call setChoiceMode() with the CHOICE_MODE_MULTIPLE_MODAL argument
Popup menus

• Instantiate a PopupMenu with its constructor
  — Current application Context and the View to which the menu should be anchored as parameters
• Use MenuInflater to inflate your menu resource into the Menu object returned by PopupMenu.getMenu()
• Call PopupMenu.show()

```xml
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_overflow"
    android:contentDescription="@string/ic_overflow"
    android:onClick="showPopup" />
```

```java
public void showPopup(View v) {
    PopupMenu popup = new PopupMenu(this, v);
    MenuInflater inflater = popup.getMenuInflater();
    inflater.inflate(R.menu.actions, popup.getMenu());
    popup.show();
}
```
Another example

```java
public void showMenu(View v) {
    PopupMenu popup = new PopupMenu(this, v);

    // This activity implements OnMenuItemClickListener
    popup.setOnMenuItemClickListener(this);
    popup.inflate(R.menu.actions);
    popup.show();
}

@Override
public boolean onMenuItemClick(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.archive:
            archive(item);
            return true;
        case R.id.delete:
            delete(item);
            return true;
        default:
            return false;
    }
}
```
Multiple devices
Multiple devices

- Use `wrap_content` and `match_parent`
- Use `RelativeLayout`
- Use configuration qualifiers
  - Create a new directory in your project's `res/` and name it using the format: `<resources_name>-<qualifier>`
    - `<resources_name>` is the standard resource name (such as `drawable` or `layout`)
    - `<qualifier>` is a configuration qualifier specifying the configuration for which these resources are to be used (such as `hdpi` or `xlarge`)
<table>
<thead>
<tr>
<th>Screen characteristic</th>
<th>Qualifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>small</td>
<td>Resources for small size screens.</td>
</tr>
<tr>
<td></td>
<td>normal</td>
<td>Resources for normal size screens. (This is the baseline size.)</td>
</tr>
<tr>
<td></td>
<td>large</td>
<td>Resources for large size screens.</td>
</tr>
<tr>
<td></td>
<td>xlarge</td>
<td>Resources for extra-large size screens.</td>
</tr>
<tr>
<td>Density</td>
<td>1dpi</td>
<td>Resources for low-density (1dpi) screens (~120dpi).</td>
</tr>
<tr>
<td></td>
<td>mdpi</td>
<td>Resources for medium-density (mdpi) screens (~160dpi). (This is the baseline density.)</td>
</tr>
<tr>
<td></td>
<td>hdpi</td>
<td>Resources for high-density (hdpi) screens (~240dpi).</td>
</tr>
<tr>
<td></td>
<td>xhdpi</td>
<td>Resources for extra-high-density (xhdpi) screens (~320dpi).</td>
</tr>
<tr>
<td></td>
<td>xxhdpi</td>
<td>Resources for extra-extra-high-density (xxhdpi) screens (~480dpi).</td>
</tr>
<tr>
<td></td>
<td>xxxhdpi</td>
<td>Resources for extra-extra-extra-high-density (xxxhdpi) uses (~640dpi). Use this for the launcher icon only, see note above.</td>
</tr>
<tr>
<td></td>
<td>nodpi</td>
<td>Resources for all densities. These are density-independent resources. The system does not scale resources tagged with this qualifier, regardless of the current screen's density.</td>
</tr>
<tr>
<td></td>
<td>tvdpi</td>
<td>Resources for screens somewhere between mdpi and hdpi; approximately 213dpi. This is not considered a &quot;primary&quot; density group. It is mostly intended for televisions and most apps shouldn't need it—providing mdpi and hdpi resources is sufficient for most apps and the system will scale them as appropriate. If you find it necessary to provide tvdpi resources, you should size them at a factor of 1.33*mdpi. For example, a 100px x 100px image for mdpi screens should be 133px x 133px for tvdpi.</td>
</tr>
<tr>
<td>Orientation</td>
<td>land</td>
<td>Resources for screens in the landscape orientation (wide aspect ratio).</td>
</tr>
<tr>
<td></td>
<td>port</td>
<td>Resources for screens in the portrait orientation (tall aspect ratio).</td>
</tr>
<tr>
<td>Aspect ratio</td>
<td>long</td>
<td>Resources for screens that have a significantly taller or wider aspect ratio (when in portrait or landscape orientation, respectively) than the baseline screen configuration.</td>
</tr>
<tr>
<td></td>
<td>notlong</td>
<td>Resources for use screens that have an aspect ratio that is similar to the baseline screen configuration.</td>
</tr>
</tbody>
</table>
Density

• \( \text{px} = \text{dp} \times (\text{dpi} / 160) \)

• For example, Android says that Launcher icons on a mobile device must be 48x48 dp.
Alternative way

- Screen configuration | Qualifier  | Examples            |
- Smallest width       | sw<N>dp    | sw600dp             |
- Available screen width | w<N>dp    | w720dp              |
- Available screen height | h<N>dp    | h1024dp             |

• typical screen widths:
  – 320dp: a typical phone screen (240x320 ldp, 320x480 mdpi, 480x800 hdpi, etc)
  – 480dp: a tweener tablet like the Streak (480x800 mdpi)
  – 600dp: a 7” tablet (600x1024 mdpi)
  – 720dp: a 10” tablet (720x1280 mdpi, 800x1280 mdpi, etc)
Example

res/layout/main.xml

```xml
<LinearLayout ...
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="luc.polimi.app.Fragment1"
        android:layout_width="match_parent" />
</LinearLayout>
```

res/layout-large/main.xml

```xml
<LinearLayout ...
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal">
    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="luc.polimi.app.Fragment1"
        android:layout_width="400dp"
        android:layout_marginRight="10dp" />
    <fragment android:id="@+id/article"
        android:layout_height="fill_parent"
        android:name="luc.polimi.app.Fragment2"
        android:layout_width="fill_parent" />
</LinearLayout>
```

res/layout-sw600dp/main.xml

```xml
<LinearLayout ...
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment android:id="@+id/headlines"
        android:layout_height="fill_parent"
        android:name="luc.polimi.app.Fragment1"
        android:layout_width="match_parent" />
</LinearLayout>
```
Different options

- Each layout can also be defined in an XML file in the `res/layout`
  - Layout aliases are then used to assign them to each configuration

res/values/layouts.xml

```xml
<resources>
  <item name="main_layout" type="layout">@layout/onepane_with_bar</item>
  <bool name="has_two_panes">false</bool>
</resources>
```

res/values-sw600dp-land/layouts.xml

```xml
<resources>
  <item name="main_layout" type="layout">@layout/twopanes</item>
  <bool name="has_two_panes">true</bool>
</resources>
```
**Manifest and devices**

- We must also declare in the manifest file which screens your application supports
  - Through `<supports-screens>` manifest element
  - If your application supports all screen sizes supported by Android (as small as 426dp x 320dp), then you don't need to declare this attribute, because the smallest width your application requires is the smallest possible on any device
Bitmaps
Bipmaps

- The system uses any size- or density-specific resources from your application and displays them without scaling
  - If resources are not available in the correct density, the system loads the default resources and scales them up or down as needed
- The system assumes that default resources (those from a directory without configuration qualifiers) are designed for the baseline screen density (mdpi)
  - A bitmap designed at 50x50 pixels for an mdpi screen is scaled to 75x75 pixels on an hdpi screen (if there is no alternative resource for hdpi)
Guidelines

• Use `wrap_content`, `match_parent`, or the `dp` unit for layout dimensions
• Do not use hard-coded pixel values in your application code
• Do not use `AbsoluteLayout` (deprecated)
• Use size and density-specific resources
Example configuration

```
res/layout/my_layout.xml          // layout for normal screen size ("default")
res/layout-large/my_layout.xml   // layout for large screen size
res/layout-xlarge/my_layout.xml  // layout for extra-large screen size
res/layout-xlarge-land/my_layout.xml  // layout for extra-large in landscape orientation

res/drawable-mdpi/graphic.png    // bitmap for medium-density
res/drawable-hdpi/graphic.png    // bitmap for high-density
res/drawable-xhdpi/graphic.png   // bitmap for extra-high-density
res/drawable-xxhdpi/graphic.png  // bitmap for extra-extra-high-density

res/mipmap-mdpi/my_icon.png      // launcher icon for medium-density
res/mipmap-hdpi/my_icon.png      // launcher icon for high-density
res/mipmap-xhdpi/my_icon.png     // launcher icon for extra-high-density
res/mipmap-xxhdpi/my_icon.png    // launcher icon for extra-extra-high-density
res/mipmap-xxxxhdpi/my_icon.png  // launcher icon for extra-extra-extra-high-density
```
New problem
Fragments

- Introduced in Android 3.0 (API level 11) to support more dynamic and flexible UI designs on large screens
- One can combine multiple fragments in a single activity to build a multi-pane UI and reuse the same fragments in multiple activities
- Represent behaviors or portions of user interface in Activities
  - Must always be embedded in an activity and their lifecycle is directly affected by the one of the host activity
  - Can be manipulated independently
- We may also use
  - A fragment without UI as invisible worker for the activity
  - Special-purpose fragments: DialogFragment, ListFragment, PreferenceFragment
Reuse

• You should design each fragment as a modular and reusable activity component
  — Each fragment defines its own layout and its own behavior
• You can include one fragment in multiple activities, so you should design for reuse
  — Avoid directly manipulating one fragment from another fragment
  — A modular fragment allows you to change your fragment combinations for different screen sizes
Fragment

- `onCreate()` initializes essential components of the fragment that you want to retain when the fragment is paused or stopped, then resumed.

- `onCreateView()` called when it's time for the fragment to draw its user interface for the first time.
  - It returns a View that is the root of the fragment's layout.
  - It can return null if the fragment does not provide a UI.

- `onPause()` called when the user is leaving the fragment.
  - This is usually where you should commit any changes that should be persisted.

- A fragment can contribute menu items to the activity's Options Menu (and, consequently, the app bar).
Activities and fragments

- Lifecycle of the activity in which the fragment lives directly affects the lifecycle of fragment
- Some further callbacks
  - Each lifecycle callback for the activity results in a similar callback for each fragment
  - `onAttach()` called when fragment has been associated with activity
  - `onActivityCreated()` called when activity's `onCreate()` method has returned
  - `onDestroyView()` called when the view hierarchy associated with fragment is being removed
  - `onDetach()` is called when the fragment is being disassociated from activity
public static class ExampleFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return infl.inflate(R.layout.ex_frag, container, false);
    }
}
Adding fragments to activities

<LinearLayout xmlns:android="...
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:baselineAligned="false">
    <fragment android:name="it.polimi.first.ArticleListFragment"
        android:id="@+id/list"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
    <fragment android:name="it.polimi.first.ArticleReaderFragment"
        android:id="@+id/viewer"
        android:layout_weight="2"
        android:layout_width="0dp"
        android:layout_height="match_parent" />
</LinearLayout>
FragmentManager

• Needed to Manage fragments in an activity
• getFragmentManager() provides a manager that can then be used to
  – Add, remove, replace fragments (through transactions)
  – Get fragments that exist in the activity
  – Pop fragments off the back stack
  – Register a listener for changes to the back stack
Adding a fragment

To add a fragment without a UI, add the fragment using `add(Fragment, String)`
- Supply a unique string "tag", instead of a view ID
- `onCreateView()` is not called

```java
FragmentManager fragmentManager = getFragmentManager();
FragmentManager fragmentTransaction = fragmentManager.beginTransaction();

ExampleFragment fragment = new ExampleFragment();
fragmentTransaction.add(R.id.fragment_container, fragment);
fragmentTransaction.commit();
```
```
// Create new fragment and transaction
Fragment newFragment = new ExampleFragment();
FragmentTransaction transaction = 
    getFragmentManager().beginTransaction();

// Replace whatever is in the fragment_container view with
// this fragment, and add the transaction to the back stack
transaction.replace(R.id.fragment_container, newFragment); 
transaction.addToBackStack(null);

// Commit the transaction
transaction.commit();
```
Fragments and Activities

• A given instance of a fragment is directly tied to the activity that contains it
  – The fragment can then access the activity instance with `getActivity()` and easily perform tasks such as find a view in the activity

• Likewise the activity can call methods in the fragment by acquiring a reference to the fragment from the FragmentManager