Mobile Application Development (COMP2008)

Lecture 2: Adaptive UIs and Activities

Updated: 9th August, 2018

Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

(Windows 1.0.1 demo slideshow.)

Outline

User Interfaces

- UIs are deceptively complex.
  - (Seeing all the options in Android Studio hints at this complexity.)
- There are many types of UI elements.
  - Each has many properties.
- But the complexity is mostly due to layouts – *where* things go.

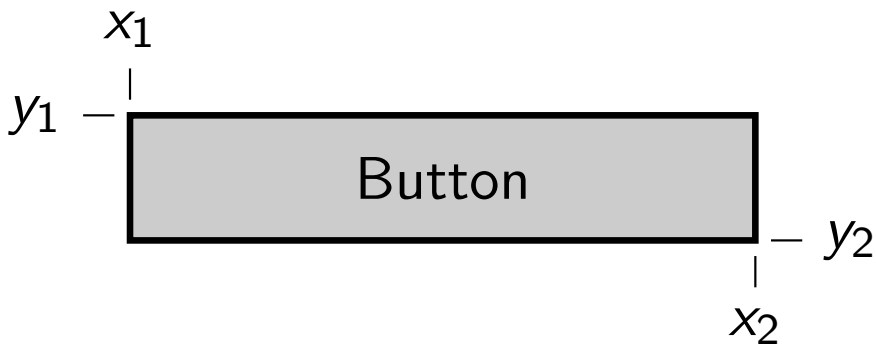
Containers and Child Elements

- GUIs (including mobile UIs) are hierarchical – they're trees[1].
- They may *look* just like a rectangle with stuff in it...
- ...But there are groups of UI elements, groups within groups, and so on.
- In Android:
  - `Button`, `EditText`, `ViewText`, etc. are kinds of "`View`".
  - "`ViewGroup`" is also a kind of `View` that contains other `Views`.
- In iOS:
  - `UIButton`, `UITextField`, etc. are kinds of "`UIView`"s.
  - Any `UIView` can contain other `UIViews` (but only some actually do, in practice).
- All UI elements occupy a rectangular area of the screen.
- Container elements are rectangles that "fence in" their child elements.

[1] The Composite Pattern, for those who've done OOSE.

Layouts

- All UI elements are rectangles, so it takes 4 numbers to represent their size and position:

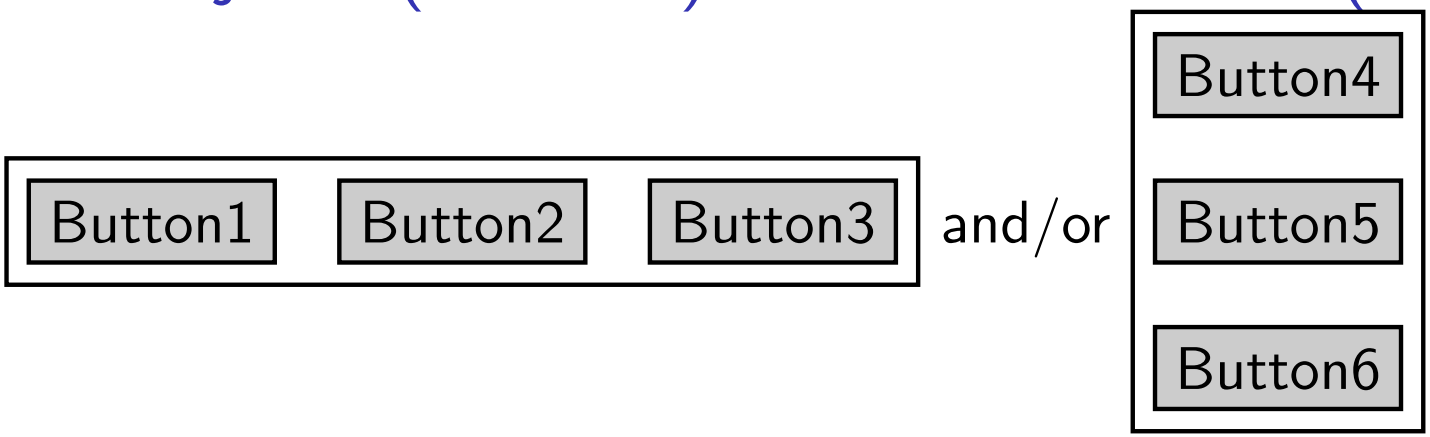


- *But* UI elements must also *adapt* to different screen sizes!
  - Different devices, rotated devices, split screen, etc.
  - x_1, y_1, x_2 and y_2 cannot be fixed at compile time.
  - They must be calculated when the GUI is displayed.
- Container elements "decide" how their child elements are sized and positioned within them.

Layouts

- In Android:
  - Different subclasses of `ViewGroup` do this differently.
  - `LinearLayout` is a container that arranges its elements in a line, horizontally or vertically.
  - `ConstraintLayout` is a container that places elements relative to each other, based on a set of "constraints" that you specify.
- In iOS:
  - `UIStackView` is comparable to Android's `LinearLayout`.
  - The "Auto Layout" feature achieves something like `ConstraintLayout`, but isn't directly tied to the view hierarchy.

LinearLayout (Android) and UIStackView (iOS)



- Horizontal/vertical stacking of UI elements – simple and powerful.
- They can be nested – you can put a horizontal panel inside a vertical one, and vice versa.
- Size and "padding" can be adjusted to fit the available space.
- The simplicity makes it possible to do UI design in XML.
  - Drag-and-drop may be easier in one sense, but editing XML gives you greater control.

[1] https://developer.android.com/guide/topics/ui/layout/linear

ConstraintLayout (Android) and Auto Layout (iOS)

- Basically designed to make the drag-and-drop editor work.
  - If you drag a button onto the screen, how does the editor define its position?
  - Using Cartesian coordinates (x, y) is really bad, because they cannot adapt to different screen sizes.
- So, we use *constraints* to define the position.
- We place each UI element *relative* to something else; e.g.
  - At the top-left of the container;
  - In the centre of the container;
  - To the right of another element (that has already been positioned).
- The drag-and-drop editor shows these constraints visually, and lets you create/delete them.

[1] https://developer.android.com/training/constraint-layout

Alternate UI Layouts

- Flexible layouts (linear, constraint-based, or others) help make adaptable UIs.
- Basically, a single layout can be stretched or squashed to fit different screens.
- But there are limits to this.
  - A big/complex layout may simply not fit on a small screen.
  - A small/simple layout may waste space on a large screen.
- It's possible (and often good!) to have alternative layouts; e.g.:
  - One for watches, one for phones, one for tablets, one for TVs; or
  - One for portrait, one for landscape.

Alternate UI Layouts: Android

Android automatically selects between different layouts as follows:

- `app/src/main/res/layout/` contains the "default" UI layout XML file(s).
- An alternate set of XML files can exist in `app/src/main/res/layout-`*qualifier*`/`.
- Where "*qualifier*" could be, for instance:

  `large` for tablet-sized devices or larger;
  `sw600dp` for screens whose width *and* height are each at least 600 "dp" units;
  `land` for landscape orientation;
  `notouch` for non-touch screens;

  ... and many others[2].
- Combos are possible; e.g. `.../res/layout-sw450dp-land`.

[1] https://developer.android.com/training/multiscreen/screensizes
[2] https://developer.android.com/guide/topics/resources/providing-resources

Side Note: Pixels and Dp

- In the past, we measured sizes and positions of GUI elements in *numbers of pixels*.
  - Most obvious way of doing it; they're *made* of pixels after all.
- But pixel density ("dots per inch") can now vary a lot between devices.
  - Once device might cram 10 times as many pixels, across the same distance, as another.
- Instead, we should use physical distance: mm, inches, etc.
- This also includes Android's "dp" (density-independent pixel) units:
  - $1\text{dp} \equiv \frac{1}{160}\text{inches}$.
  - $\Rightarrow 1\text{dp} \approx 0.16\text{mm}$.
  - $\Rightarrow 600\text{dp} \approx 95\text{mm}$.

[2] https://developer.android.com/training/multiscreen/screendensities

Behind the UI: The Activity Lifecycle

- We need to discuss some *non-UI* stuff now.
  - Because the UI has an impact on how the whole app behaves.
- An activity is the main entry point for your app, but it's not quite like a *main()* method.
- An `Activity` object goes through different stages of life:
  - Running – visible and awaiting user input;
  - Paused – still visible but not in control right now;
  - Stopped – not currently shown to the user;
  - Destroyed – deallocated from memory.

[2] https://developer.android.com/guide/components/activities/activity-lifecycle

Activity State Transitions

- There are various transitions between *running*, *paused*, *stopped* and *destroyed*.
- `onCreate()` (the most important), `onStart()`, `onResume()`, `onPause()`, `onStop()`, `onDestroy()`.
  - All overridable methods from the `Activity` class.
- Surprisingly (perhaps), activities may be destroyed and re-created when the *screen orientation* changes.
- A stopped activity may also be destroyed if the currently-running activity needs more memory.

Surviving Destruction: Saving and Reloading State

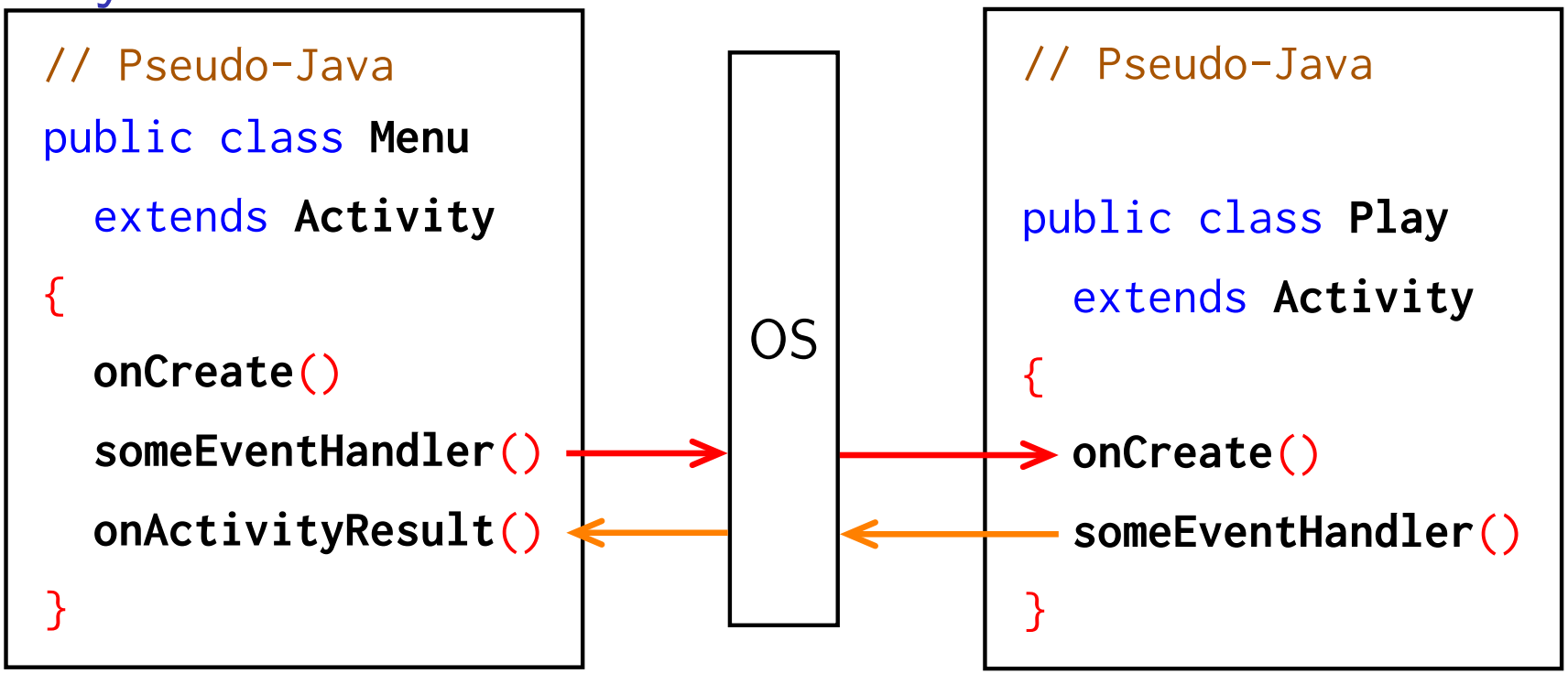
- Prior to destruction, the `onSaveInstanceState(Bundle)` method is called.
- On (re-)creation, `onCreate(Bundle)` is called.
- A `Bundle` object[3] is an arbitrary-stuff-container.
  - It's a map, storing key-value pairs.
  - The keys are strings, and the values are a range of types (primitives, arrays and classes).
- In `onSaveInstanceState()`, you save any important activity data to the provided Bundle.
- In `onCreate()`, you get back the same Bundle, so you can reconstruct the original state of the activity.

[3] https://developer.android.com/reference/android/os/Bundle

Multiple Activities

- An activity basically controls one kind of UI.
  - (There could be multiple UI layouts, but different layouts are still essentially the same UI.)
- Many/most apps need more than one. e.g. for a mobile game app, you could have activities for:
  - Showing the gameplay itself;
  - Selecting which level to play next;
  - Managing settings (like sound, difficulty, etc.)
- Starting activities is the OS's responsibility, but we can tell it to do so.

Activity Communication



1. Menu (in an event handler) sends a message to the OS.
2. The OS instantiates Play, and calls Play.onCreate().
3. Play (in an event handler) sends a return result to the OS.
4. The OS calls Menu.onActivityResult().

- (Steps 3 and 4 are optional.)

Starting an Activity

```
// Within class Menu.
playButton.setOnClickListener(new View.setOnClickListener()
{
    @Override public void onClick(View v)
    {
        startActivity(new Intent(Menu.this, Play.class));
    }
});
```

- When the button is pressed, we call `startActivity()` to send an "intent" to the OS.
- The `Intent` object is a message, indicating which activity to start.
  - `Menu.this` is the existing Activity object.
  - `Play.class` gives the class of the new Activity.

[3] https://developer.android.com/training/basics/firstapp/starting-activity

Passing Data Between Activities

- To pass data to the new activity, add some "extras" to the intent object:

```
// In Menu (the calling activity), on button press
Intent intent = new Intent(Menu.this, Play.class);
intent.putExtra("lives", 5);
intent.putExtra("level", "Overworld");
startActivity(intent);
```

- "Extras" works like `Bundle` (because it is, internally).
- The receiving activity (`class PlayActivity`) can then query the intent that created it:

```
// In Play.onCreate() (the new activity)
Intent intent = getIntent();
int nLives = intent.getIntExtra("lives");
String level = intent.getStringExtra("level");
```

Starting Activities: Good Practice

- Let an activity class provide an intent for itself.
- Give it constants for the extra keys.
- Avoid key naming conflicts by prefixing the package name:

```
public class Play extends Activity
{
    private static final String LIVES = "com.example.lives";
    private static final String LEVEL = "com.example.level";

    public static Intent getIntent(Context c,
                                   int lives, String level)
    {
        Intent intent = new Intent(c, PlayActivity.class);
        intent.putExtra(LIVES, lives);
        intent.putExtra(LEVEL, level);
        return intent;
    }
    ...
```

Starting Activities: Good Practice

- This way we keep all the data bundling/unbundling in one place.
  - Only *one* activity needs to know about it.
  - Limits coupling between the activities. (Recall from ISE that we prefer low coupling!)
- Things are now simpler on the calling side:

```
// In Menu (the calling activity), on button press
startActivity(Play.getIntent(Menu.this, 5, "Overworld"));
```

Sending Back Results (1)

- Often a new (child) activity is used to *return a result* to its caller.
- A result has three parts:
  1. A "request code" (integer) indicating *which* child activity (in case there are several).
  2. A "result code", generally one of the integer constants `RESULT_OK` or `RESULT_CANCELLED`.
  3. An `Intent` object, which can contain arbitrary "extras" data.
- First, when *starting* an activity that returns a result, we must pick a request code ourselves:

```
// In Menu (the calling activity)
private static final int REQUEST_CODE_PLAY = 0;
...
Intent intent = PlayActivity.getIntent(...);
startActivityForResult(intent, REQUEST_CODE_PLAY);
```

Sending Back Results (2)

- The "back" button automatically sends RESULT_CANCELLED.
- We can make whatever event we like send back RESULT_OK.

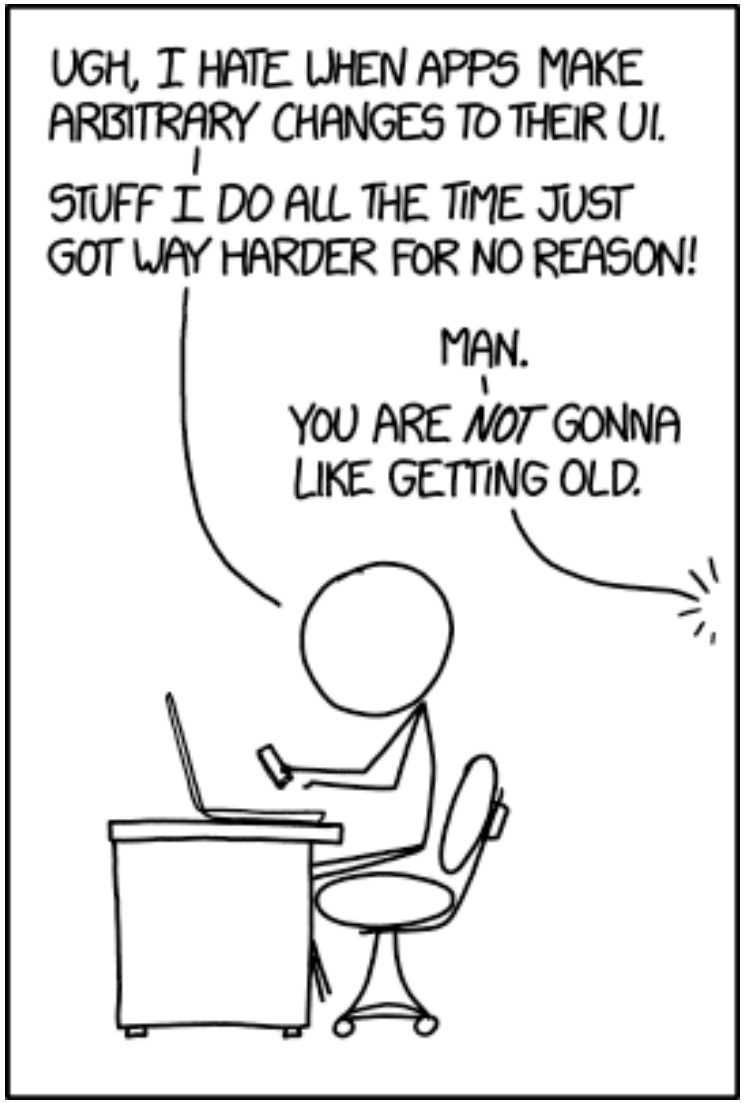
```
// In Play (the activity being called)
private static final int EXTRA_SCORE = "com.example.score";
private int gameScore = 0; // Updated while game runs.
...
// Inside an event handler:
Intent returnData = new Intent(); // The return intent
returnData.putExtra(EXTRA_SCORE, gameScore);
setResult(RESULT_OK, returnData);
...
// To allow the caller to retrieve the data:
public static int getScore(Intent intent)
{
    return intent.getIntExtra(EXTRA_SCORE);
}
```

Sending Back Results (3)

- The calling activity receives the result:

```
// In Menu (the caller, again)
@Override
protected void onActivityResult(int requestCode,
                                int resultCode,
                                Intent returnData)
{
    if(resultCode == RESULT_OK &&
       requestCode == REQUEST_CODE_PLAY)
    {
        score = PlayActivity.getScore(returnData);
    }
    else if(other things happened) {...} // If needed
}
```

- Once again, we make the child responsible for both bundling and unbundling the intent data.



(https://xkcd.com/1770/)