Android Application Programming

Challenge: Changing the Orientation and Saving State

Introduction

Your tic-tac-toe game only works in portrait mode. If you were to flip the emulator into landscape mode by pressing Ctrl-F11, you would see the board is cleared of all the images, and the text beneath the board is not visible. While we could force the user to only use our app in portrait mode, we should make our app more flexible by allowing the user to play with it in either mode.

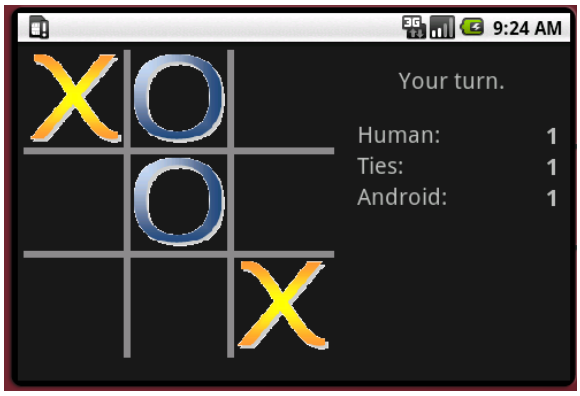
The goal of this tutorial is to show you how to develop an application that can work in portrait and landscape mode. You will also learn how to make the application's data persist when the orientation is changed and how to make data persist between application invocations.

Changing the Orientation

When you press Ctrl-F11, your emulator will switch from portrait mode to landscape mode. Pressing Crtl-F11 again switches it back. Your game was built for portrait mode, so it does not behave properly when switched to landscape mode.

You can force Android to only show your app in portrait mode: just add `android:screenOrientation="portrait"` (or `"landscape"` if you wish) to the `<activity>` element in the app's AndroidManifext.xml file. However, best practice is to allow the user to interact with the app in either orientation.

Our goal is to create a landscaped orientation that looks like the screenshot on the right.



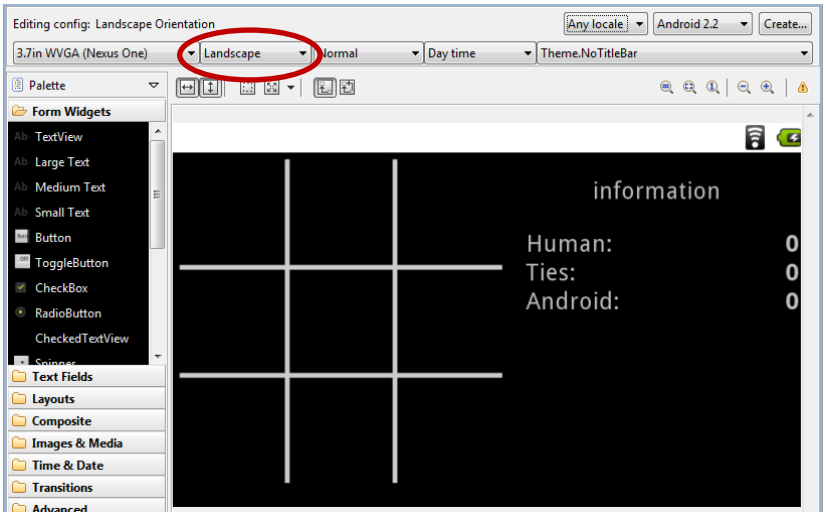
1. To create more screen real estate, remove the title bar from the application window by modifying the `application` element in `AndroidManifest.xml`:

```
<application
    android:icon="@drawable/icon"
    android:label="@string/app_name"
    android:theme="@android:style/Theme.NoTitleBar" >
```

2. Create a `res/layout-land` directory.

3. Copy the `res/layout/main.xml` file into the `layout-land` directory. This is the file that Android will apply to the Activity's View when the emulator is put in landscape mode.

4. Open the `layout-land/main.xml` file and change the size of the `BoardView` to 270x270 dp to make it take up less space. Also align it on the left side of the screen. To do this, you can change the `LinearLayout`'s `orientation` property to `horizontal` or replace the `LinearLayout` entirely with a `RelativeLayout`. You will also need to change the layout of the `TextView` controls so they are displayed to the right of the game board. **It is left to you to make these edits.** To test your layout without running your game in the emulator, you can switch to Graphical Layout mode and set the orientation to Landscape as shown below.

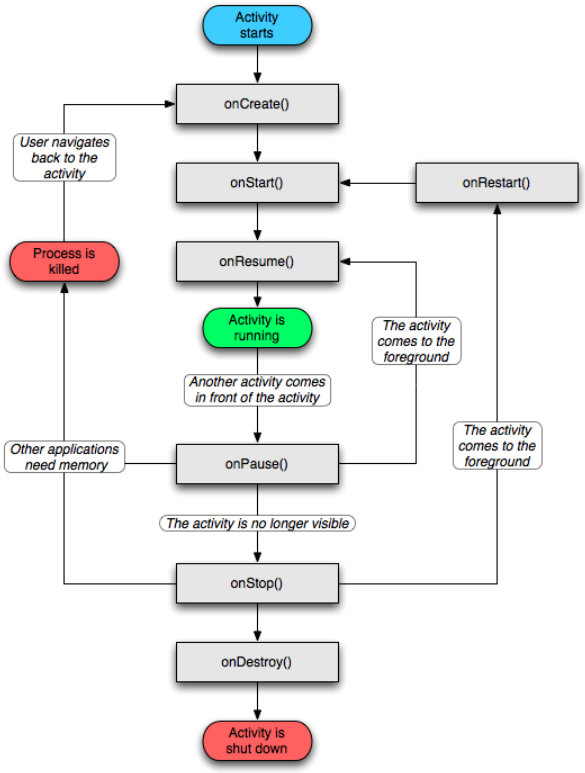


5. After you have the `BoardView` and `TextViews` aligned properly, run your program and press Ctrl-F11 to change its orientation. Your app should switch orientations and display all the screen elements correctly.

   Try making a few moves, and before the game is over, press Ctrl-F11. What happens? When the orientation is changed in the middle of the game, the board is cleared and your game starts over from the beginning. If you play a few games and then change the orientation, you'll notice that the scores are also reset to zero. You can imagine how frustrated a user would get if they were about to win a game, they shifted their phone a little which caused the orientation to change, and the game reset! Ideally we'd like our game to maintain its state when changing orientation, so we'll fix this problem next.

Saving Instance State

Before discussing how state information is stored, it is helpful to get a firm grasp on the life cycle of an Activity. The Activity life cycle is illustrated in the diagram[1] below. Each rectangle represents the Activity's methods which are called in response to different events. For example, after starting an Activity, the `onCreate()` method is called first followed by `onStart()` and `onResume()`. The application then runs until another Activity comes in front of the currently running Activity, causing the `onPause()` method to trigger. If the Activity is no longer visible, `onStop()` triggers, and if Android decides another process needs the memory, the Activity is killed. However, if the Activity is navigated to before being killed, `onRestart()` is triggered followed by `onStart()` and `onResume()`.



Changing the orientation of an Android device will cause your Activity to terminate and restart, so all the app's

[1] Figure is from *Application Fundamentals* at http://developer.android.com/guide/topics/fundamentals.html

variables are reset, and the game is started anew. This is just like what happens when when your Activity is no longer in the foreground, its memory is reclaimed, and you navigate back to the app.

Fortunately, Android provides a mechanism to remember whatever pieces of information we would like when the Activity is started again after changing the device's orientation. We first need to decide what information is necessary to start our game back in the same state. We'll bundle the information together before the Activity is killed, and Android will pass it back to us in the `onCreate()` method. We'll extract the bundled information and put it back into the necessary variables to pick up right where we left off.

1. To save the state of the game, you need to override the `onSaveInstanceState()` method for the `AndroidTicTacToe` class (the Activity). This method is called by Android before it pauses the application (before `onPause()` is triggered), and it passes the method a `Bundle` object which can be used to save key/value pairs that will be given back to the application (via another call to `onCreate()`) even if the Activity is dropped from memory.

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    outState.putCharArray("board", mGame.getBoardState());
    outState.putBoolean("mGameOver", mGameOver);
    outState.putInt("mHumanWins", Integer.valueOf(mHumanWins));
    outState.putInt("mComputerWins", Integer.valueOf(mComputerWins));
    outState.putInt("mTies", Integer.valueOf(mTies));
    outState.putCharSequence("info", mInfoTextView.getText());
    outState.putChar("mGoFirst", mGoFirst);
}
```

   For the above code to work, you will need to create a `getBoardState()` function for the `TicTacToeGame` class which returns a character array representing the state of the board. **It is left for you to implement this method.**

2. When `onCreate()` is called the first time, its parameter `savedInstanceState` will be `null`, and the game should be started like usual. But after changing the orientation, `onCreate`'s `savedInstanceState` will contain the key/value pairs placed in `onSaveInstanceState`'s `outState` bundle. You will need to extract the values from the `Bundle` and place them back into their associated variables like so:

```
@Override
public void onCreate(Bundle savedInstanceState) {

skip...

    if (savedInstanceState == null) {
        startNewGame();
    }
    else {
        // Restore the game's state
        mGame.setBoardState(savedInstanceState.getCharArray("board"));
        mGameOver = savedInstanceState.getBoolean("mGameOver");
        mInfoTextView.setText(savedInstanceState.getCharSequence("info"));
        mHumanWins = savedInstanceState.getInt("mHumanWins");
        mComputerWins = savedInstanceState.getInt("mComputerWins");
        mTies = savedInstanceState.getInt("mTies");
        mGoFirst = savedInstanceState.getChar("mGoFirst");
    }
    displayScores();
}
```

For the above code to work, you will need to create a `setBoardState()` function for the `TicTacToeGame` class that allows the board state to be set to the values in the given char array. You will likely find the array's `clone()` method to be helpful. **This is left to you to complete.**

3. The code above also calls a `displayScores()` function to show the scores. Declare it like so:

```
private void displayScores() {
    mHumanScoreTextView.setText(Integer.toString(mHumanWins));
    mComputerScoreTextView.setText(Integer.toString(mComputerWins));
    mTieScoreTextView.setText(Integer.toString(mTies));
}
```

4. Note that there is also an `onRestoreInstanceState()` method that can be overridden for the Activity to restore the application's state. So instead of placing your restore code in the else statement in step 2, you could do this:

```
@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);

    mGame.setBoardState(savedInstanceState.getCharArray("board"));
    mGameOver = savedInstanceState.getBoolean("mGameOver");
    mInfoTextView.setText(savedInstanceState.getCharSequence("info"));
    mHumanWins = savedInstanceState.getInt("mHumanWins");
    mComputerWins = savedInstanceState.getInt("mComputerWins");
    mTies = savedInstanceState.getInt("mTies");
    mGoFirst = savedInstanceState.getChar("mGoFirst");
}
```

5. Run your application and make some moves. When you press Ctrl-F11, the board should remain in the same condition it was before. However, you'll probably notice that the computer makes an additional move when it's your turn! This is because the variable that ensures it's the correct person's turn is not being saved. **It's left to you to fix this bug.**

Saving Persistent Information

If you were to click the Back button on the emulator and restart the application, either by clicking on the app's icon or using Eclipse to restart, your application's state is not saved; your game board will be cleared and your game scores (Human, Tie, Android) will be reset to 0. That's because Android only saves state information (by calling `onSaveInstanceState()`) if *Android* is responsible for killing the Activity. If *you* kill it by clicking the Back button, for example, then `onSaveInstanceState()` is not called.

In order for information to persist between application restarts, there are several mechanisms to choose from. You could store data to a file and read it back in, or you could save data to a SQLite database. If you only need to store simple key/value pairs like we have been doing, the `SharedPreferences` is the ideal class to use.

Follow the instructions below to use the `SharedPreferences` class to make the game scores persist between application restarts:

1. Declare a SharedPreferences data member for the AndroidTicTacToe class:

```
private SharedPreferences mPrefs;
```

2. In the onCreate() method, use Activity.getSharedPreferences() to initialize mPrefs. The first argument is the name of the preference file, and the second argument is typically MODE_PRIVATE or 0.

```
mPrefs = getSharedPreferences("ttt_prefs", MODE_PRIVATE);
```

3. When your application is being terminated, the Activity's onStop() method will be called (refer back to the Activity life cycle figure shown earlier). This is your chance to save any persistent information. Use the SharedPreferences object to save the scores like so:

```
@Override
protected void onStop() {
   super.onStop();

   // Save the current scores
   SharedPreferences.Editor ed = mPrefs.edit();
   ed.putInt("mHumanWins", mHumanWins);
   ed.putInt("mComputerWins", mComputerWins);
   ed.putInt("mTies", mTies);
   ed.commit();
}
```

Note that the SharedPreferences.Editor has methods for storing any primitive type. Calling commit() replaces any of the data that was previously stored in SharedPreferences with the new data.

4. Now we need to restore the game scores from mPrefs when the onCreate() method is being called:

```
mPrefs = getSharedPreferences("ttt_prefs", MODE_PRIVATE);

// Restore the scores
mHumanWins = mPrefs.getInt("mHumanWins", 0);
mComputerWins = mPrefs.getInt("mComputerWins", 0);
mTies = mPrefs.getInt("mTies", 0);
```

In the code above, getInt() is supplied 0 for the second argument which will be the value returned if the preferences have not been previously saved (like the first time you run your modified application).

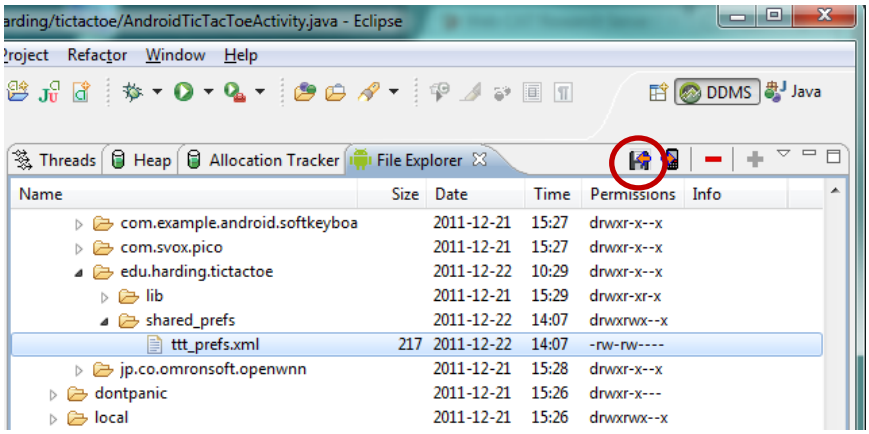
5. Your app will now save the scores indefinitely. However, the user may not want to be reminded that Android beat him 20 times at tic-tac-toe. So let's give the user the ability to reset the scores. Add a new menu item to res/menu/options_menu.xml labeled "Reset Scores", and add an image for the menu item. To keep the menu from getting cluttered, remove the Quit option which really isn't necessary anyway.

6. When the Reset Scores menu item is selected, set the mHumanWins, mComputerWins, and mTies variables to 0 and re-display the scores.

7. Since we are saving the values of mHumanWins, mComputerWins, and mTies in the preferences, you can safely remove all the code you previously entered that saved their values with savedInstanceState.

Now run your application and play a few games. Take a look at the scores. Click the Back button on the emulator and restart your game by clicking on the TicTacToe icon or starting it from Eclipse. The scores should be the same as they were before you started a new game. Now test your Reset Scores menu item. The scores should immediately go back to 0.

You can see the values that were stored in your shared preferences by going back to Eclipse and opening the **DDMS perspective**. This is done by clicking the DDMS button in the upper-right corner of Eclipse. If you don't see this button, select **Window > Open Perspective > Other…** from Eclipse's menu and choose **DDMS** from the dialog box. The DDMS persepective will now be open, and you will see a DDMS button in the upper-right corner of Eclipse.



The figure above shows the **File Explorer** which was used to navigate to `data/data/edu.harding.tictactoe/shared_prefs`. Here you will find the `ttt_prefs.xml` file that was created by your app. If you would like to see the contents of the file, select the file and press the disk icon which is circled in the figure above. You can then save it using the save file dialog box that appears. The contents of the file will look something like this:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<int name="mComputerWins" value="1" />
<int name="mTies" value="0" />
<int name="mHumanWins" value="1" />
</map>
```

Extra Challenge

There are two extra challenges:

1. The game's difficulty level is not being saved, so every time the game starts, the difficulty level is reset to Expert. Fix this by saving the game difficulty level to `SharedPreferences`. This may be somewhat challenging because the game's difficulty level is using an enumerated type, and the `SharedPreferences.Editor` doesn't have a method to save the value of an enumerated type. You should save an integer which corresponds to the enumerated value and convert that int back into an enumerated type when you retrieve the preference.

2. If you press Ctrl-F11 immediately after making a move and before the computer has moved, the application will likely crash. You'll find the LogCat helpful for determining what is causing the crash. A simple try-catch block around the line of code causing the exception will stop the app from crashing. However, the game will still not work correctly when changing the orientation immediately before the computer moves; the game will indicate it's still Android's turn, but the computer will never make a move. This problem has to do with the `Handler` executing code on old instances of an `Activity` which has been terminated. You need to fix this problem by making the computer make a move after the new `Activity` has restarted if it's the computer's turn to move. This requires adding a few lines of code to the `onCreate()` method.