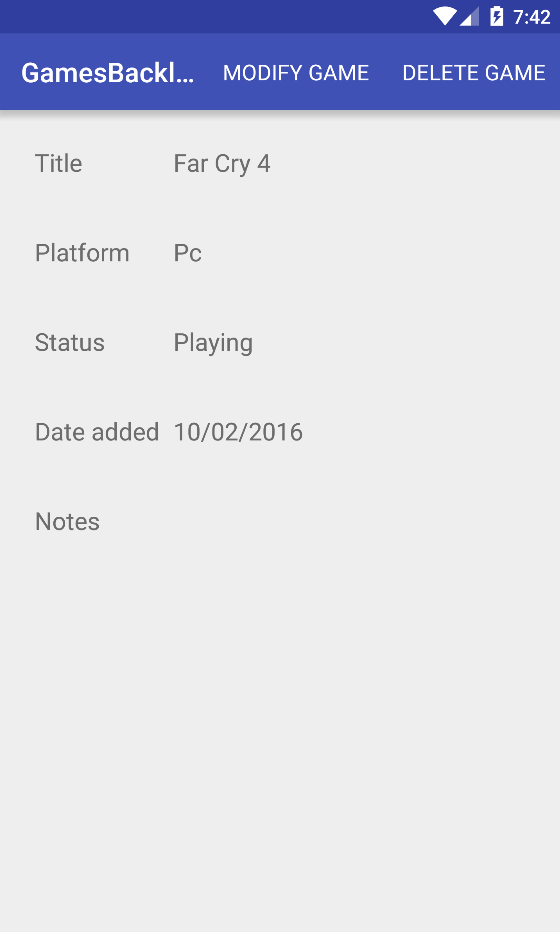
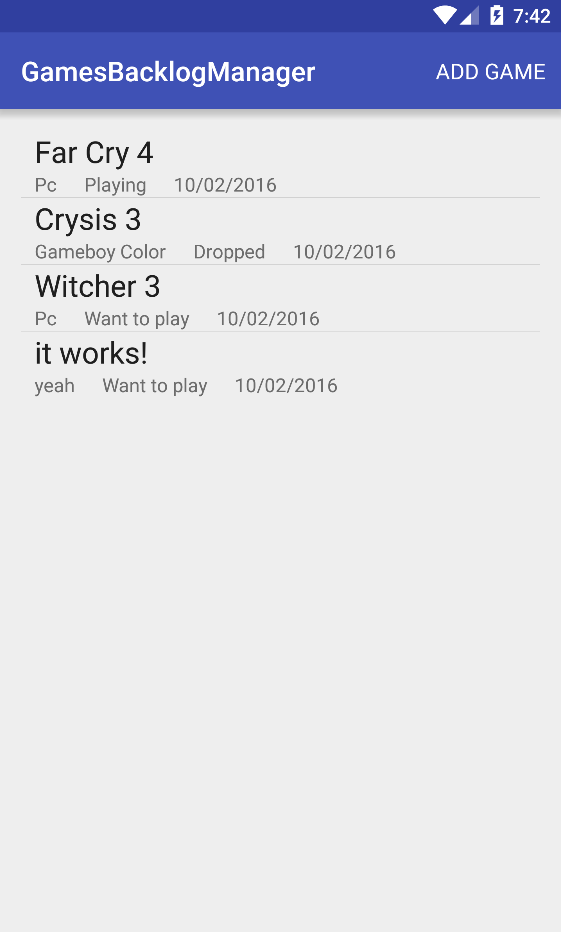
**Game Backlog**

We’re going to create an application where you can keep track of which games you have played and which games you still want to play. The application will use a ListView which will display all Games by using a custom adapter. To store the games in the application, we will use SharedPreferences, which saves simple values in a file that can be used later. We will use an external JAR named ‘gson’ to convert our games to JSON, so that we can save it in SharedPreferences.





The tutorial will provide you all the necessary steps. But if you encounter the grinding gear icon and you are expected to do it by yourself without the help of the tutorial.

# **Step 1: Setting up the project and import the gson library**

## 1.1: Create the project

# Create a new project called ‘GamesBacklogManager’ with an Empty Actvity.

# I would recommend to organize the java files in packages. That’s why we’re going to create a package called ‘activity’ by right clicking on ‘com.\*your com name here\*.gamesbacklogmanager’ in the file explorer and select new > package.

# Now we’re going to move MainActivity to package ‘activity’ by right clicking on MainActivity in the file explorer, then refactor > move, which will open up a new dialog. At ‘To Package’, click on the button with the three dots, and find and select package ‘activity’. Finally, press Refactor.

## 1.2: Add the gson library to the project

# Android Studio can use Maven repositories to import dependencies, including the gson library that we need. To do this, expand ‘Gradle Scripts’ in the file tree, and open ‘build.gradle’ of ‘(Module: app)’. Make sure you don’t accidentally open the ‘build.gradle’ of ‘(Project: GamesBacklogManager)’.

# Add this line to section ‘dependencies’:

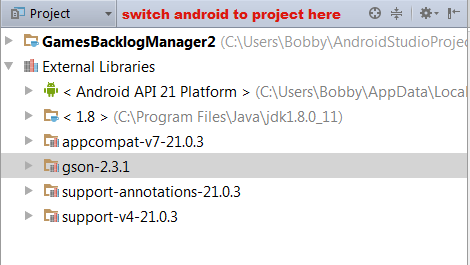
|  |
| --- |
| compile **'com.google.code.gson:gson:2.3.1'** |

This dependency makes use of the 2.3.1 version. It is possible that you have the updated version of this dependency. To check this go to Build-> Edit Libraries and Dependencies and click the plus button to add a library dependency. Search the gson one and it is added manually.

# The file should look like this:

# gradle build file.png

# After this, go to the top bar and go to Build>Rebuild Project. Rebuilding the project may take a while. After this is done, the gson library should be imported. To check this, switch the ‘Android’ file structure view to ‘Project’ (top left corner of file structure), and expand ‘External libraries’:



When done, switch back to the Android Structure.

## 1.3: Filling Strings.xml

During this assignment, a couple of Strings are used in the application, which are stored in res>values>strings.xml. Add these strings to Strings.xml:

|  |
| --- |
| <**resources**>  <**string name="app\_name"**>GamesBacklogManager</**string**>  <**string name="hello\_world"**>Hello world!</**string**>  <**string name="action\_settings"**>Settings</**string**>  <**string name="game\_backlog"**>Game Backlog</**string**>  <**string name="title\_activity\_game\_details"**>GameDetailsActivity</**string**>  <**string name="title"**>Title</**string**>  <**string name="platform"**>Platform</**string**>  <**string name="status"**>Status</**string**>  <**string name="date\_added"**>Date added</**string**>  <**string name="notes"**>Notes</**string**>  <**string name="default\_value"**>Default value</**string**>  <**string name="game\_details"**>Game Details</**string**>  <**string name="dialog\_game\_deletion\_confirmation"**>Are you sure you want to delete this game?</**string**>  <**string name="delete\_game"**>Delete game</**string**>  <**string name="modify\_game"**>Modify game</**string**>  <**string name="cancel"**>Cancel</**string**>  <**string name="game\_deleted"**>Game has been deleted</**string**>  <**string name="title\_activity\_add\_game"**>Add Game</**string**>  <**string name="save\_game"**>Save game</**string**>  <**string name="add\_game"**>Add Game</**string**>  <**string name="pick\_random\_game"**>Pick random game</**string**>  <**string name="title\_is\_required"**>Title is required!</**string**>  <**string name="platform\_is\_required"**>Platform is required!</**string**>  <**string name="title\_field\_is\_empty"**>Title field is empty</**string**>  <**string name="plaftorm\_field\_is\_empty"**>Platform field is empty</**string**>  <**string name="game\_has\_been\_added"**>Game has been added</**string**>  <**string name="title\_activity\_modify\_game"**>Modify Game</**string**>  <**string name="game\_has\_been\_modified"**>Game has been modified</**string**>  *<!--String that we are going to use for the Spinner-->*  <**string-array name="game\_status"**>  <**item**>Want to play</**item**>  <**item**>Playing</**item**>  <**item**>Stalled</**item**>  <**item**>Dropped</**item**>  </**string-array**>  </**resources**> |

# Notice that there is also a string-array present. This will be used for the Spinner, which is Android’s version of a dropdown list, when adding or modifying an activity.

# **Step 2: Create a custom adapter**

# To display games in a list, we’re going to need a few things, namely a ListView, a Game model, a row layout and a custom ArrayAdapter. Because this has been explained in previous assignments, I will not explain or show everything explicitly.

## 2.1: Create the ListView

Inside activity\_main.xml, remove the TextView, and replace it with a ListView, with id ‘gameList’:

|  |
| --- |
| <**ListView**  **android:layout\_width="wrap\_content"**  **android:layout\_height="wrap\_content"**  **android:id="@+id/gameList"**  **android:layout\_alignParentTop="true"**  **android:layout\_alignParentLeft="true"**  **android:layout\_alignParentStart="true"** /> |

## 2.2: Create the model

First, create a package called ‘model’. Inside that package, create a class called ‘Game’:

|  |
| --- |
| **public class** Game **implements** Serializable {  **private long id**;  **private** String **title**;  **private** String **platform**;  **private** Date **dateAdded**;  **private** String **gameStatus**;  **private** String **notes** = **""**;  **public** Game(){}  **public** Game(**int** id, String title, String platform, Date dateAdded, String gameStatus, String notes) {  **this**.**id** = id;  **this**.**title** = title;  **this**.**platform** = platform;  **this**.**dateAdded** = dateAdded;  **this**.**gameStatus** = gameStatus;  **this**.**notes** = notes;  }  **public long** getId(){  **return id**;  }  **public void** setId(**long** id){  **this**.**id** = id;  }  **public** String getTitle() {  **return title**;  }  **public void** setTitle(String title) {  **this**.**title** = title;  }  **public** String getPlatform() {  **return platform**;  }  **public void** setPlatform(String platform) {  **this**.**platform** = platform;  }  **public** Date getDateAdded() {  **return dateAdded**;  }  **public void** setDateAdded(Date dateAdded) {  **this**.**dateAdded** = dateAdded;  }  **public** String getGameStatus() {  **return gameStatus**;  }  **public void** setGameStatus(String gameStatus) {  **this**.**gameStatus** = gameStatus;  }  **public** String getNotes() {  **return notes**;  }  **public void** setNotes(String notes) {  **this**.**notes** = notes;  }  } |

Notice that we’re implementing the Serializable interface. This is necessary if we want to send and receive Games from and to Activities.

## 2.3: Create the row layout

Now we’re going to make the layout in which every Game will be displayed. In the layout folder inside the res folder, create a new Layout Resource File called ‘single\_game\_item.xml’.



The layout consists of multiple TextViews: one big TextView (title), and multiple small TextViews below it (platform, status, date added). Build the next layout:

# single game item.png

## 2.4: Creating the custom ArrayAdapter

# To display our Game in the list, we still need to create the custom ArrayAdapter. Create a package called ‘adapter’, and create a new java class called ‘GameListItemAdapter’ inside the package:

|  |
| --- |
| **public class** GameListItemAdapter **extends** BaseAdapter {  **private** List<Game> **gameArrayList**;  **private** Context **context**;  **private** LayoutInflater **inflater**;  **public** GameListItemAdapter(List<Game> list, Context context){  **this**.**gameArrayList** = list;  **this**.**context** = context;  **inflater** = LayoutInflater.*from*(context);  }  @Override  **public int** getCount() {  **return gameArrayList**.size();  }  @Override  **public** Game getItem(**int** position) {  **return gameArrayList**.get(position);  }  @Override  **public long** getItemId(**int** position) {  **return gameArrayList**.get(position).getId();  }  @Override  **public** View getView(**int** position, View convertView, ViewGroup parent) {  View row = convertView;  ViewHolder holder;  *//Check if the row is new*  **if** (row == **null**) {  *//Inflate the layout if it didn't exist yet*  row = **inflater**.inflate(R.layout.***single\_game\_item***, parent, **false**);  *//Create a new view holder instance*  holder = **new** ViewHolder(row);  *//set the holder as a tag so we can get it back later*  row.setTag(holder);  } **else** {  *//The row isn't new so we can reuse the view holder*  holder = (ViewHolder) row.getTag();  }  *//Populate the row*  holder.populateRow(getItem(position));  **return** row;  }  **class** ViewHolder{  **private** TextView **title**;  **private** TextView **platform**;  **private** TextView **status**;  **private** TextView **date**;  *//initialize the variables*  **public** ViewHolder(View view){  **title** = (TextView) view.findViewById(R.id.***gameTitle***);  **platform** = (TextView) view.findViewById(R.id.***gamePlatform***);  **status** = (TextView) view.findViewById(R.id.***gameStatus***);  **date** = (TextView) view.findViewById(R.id.***gameDate***);  }  **public void** populateRow(Game game){  **title**.setText(game.getTitle());  **platform**.setText(game.getPlatform());  **status**.setText(game.getGameStatus());  *//Convert Date object to String by formatting it*  SimpleDateFormat format = **new** SimpleDateFormat(**"dd/MM/yyyy"**);  String dateString = format.format(game.getDateAdded());  **date**.setText(dateString);  }  }  } |

## 2.5: Test the adapter

Now that we have everything ready, it’s possible to fill the list with games. Declare the ListView, and initialize it. Create one or more Games(pass them a number, some strings and a ‘new Date()’ object) and put them in an ArrayList. Create a GameListItemAdapter and pass it the list of games, along with the single\_game \_item layout. Finally, set the adapter for the ListView object.

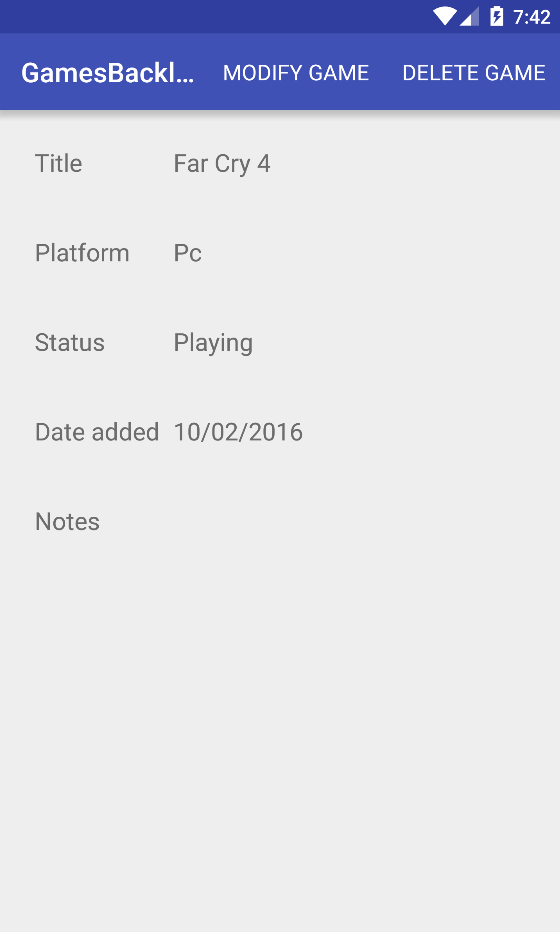
# **Step 3: Creating the details screen, and sending data to it**

When we click on a game, we want to go to the details screen, which will display the title, platform, status and the date-added again, but this time also optional notes.

## 3.1: Creating the layout



Make a detail activity with the following layout:



## 3.2: Send a game to GameDetailsActivity

Let’s send data to the details activity. Open MainActivity.java, and add the following code inside method ‘oncreate’, at the bottom of the method:

|  |
| --- |
| **gameList**.setOnItemClickListener(**new** AdapterView.OnItemClickListener() {  *//Will trigger when the user clicks on a game*  @Override  **public void** onItemClick(AdapterView<?> parent, View view, **int** position, **long** id) {  Intent intent = **new** Intent(MainActivity.**this**, GameDetailsActivity.**class**);  *//Get the correct game based on which listitem got clicked, and put it as parameter in the intent*  Game selectedGame = (Game) parent.getAdapter().getItem(position);  intent.putExtra(**"selectedGame"**, selectedGame);  *//Open GameDetailsActivity*  startActivity(intent);  }  }); |

Because we’ve implemented the Serializable interface inside Game.java, we can put a Game object inside an intent, and retrieve it in GameDetailsActivity.

## 3.3: Displaying a game in GameDetailsActivity

Open GameDetailsActivity.java, and declare a Game object outside of the methods. Do the same for five TextView objects, and call them title, platform, status, date and notes. Inside the onCreate method, initizalize the TextViews.

Now we’re going to retrieve the Game object that we’ve sent with our intent, that we’ve sent when we clicked on a list item (still inside method onCreate):

|  |
| --- |
| *//Get the game from the intent, which was passed as parameter*  **game** = (Game) getIntent().getSerializableExtra(**"selectedGame"**); |

Before we are going to set the game data into the TextViews, we first have to convert attribute ‘dateAdded’ of the game from a Date object to a String:

|  |
| --- |
| *//Get the Date object from the game, and convert it to a day/month/year String, by formatting it with a SimpleDateFormat object*  String dateString = **new** SimpleDateFormat(**"dd/MM/yyyy"**).format(**game**.getDateAdded()); |



Finally, we set the game data into the Views:Test the application to see if it works.

# **Step 4: Adding games**

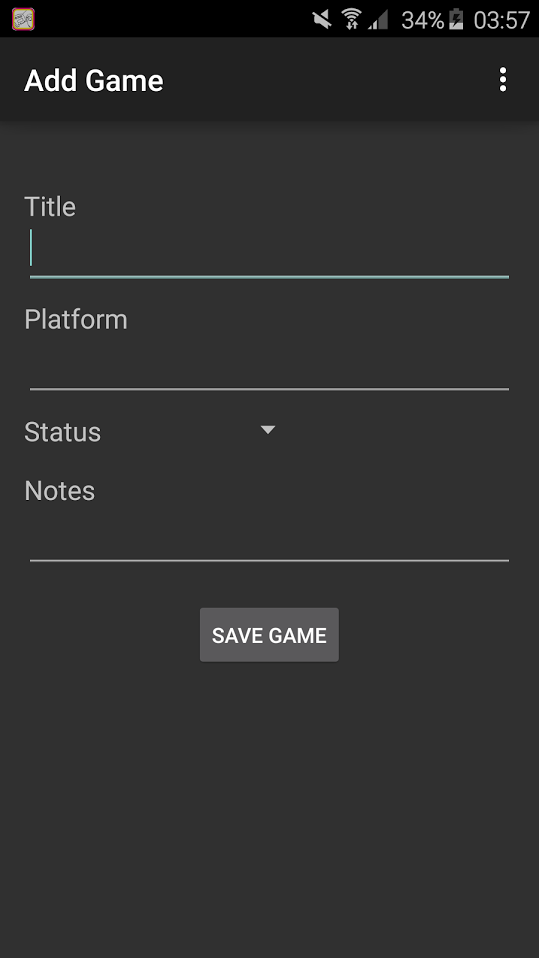
It’s time to add the functionality to add games to our backlog. We will make a new activity, where we can fill in the information about the game, which we will store in a list of games, which we will store in SharedPreferences. SharedPreferences remembers data even when the application gets closed, but it does not really accept objects, such as games. Because Strings are an exception, we will convert the list of games into a JSON String with the help of the external library called Gson.

## 4.1: Creating the layout

Create a new activity inside package activity, and call it AddGameActivity.



The activity should look like this, we will use a Spinner for the status:



## 4.2: Initializing the Views and setting the listener



Open AddGameActivity. Before we’re going to create the save functionality, we have to declare and initialize our Views. Declare the following Views outside the methods:

Next, we’re going to fill the Spinner with items. In our Strings.xml we created a string-array, which contains all the game statuses. We are going to fill the Spinner with the string-array from the .xml file by using a specific method of the ArrayAdapter class called createFromResource:

|  |
| --- |
| *// Create an ArrayAdapter using the string array and a default spinner layout*  ArrayAdapter statusAdapter = ArrayAdapter.*createFromResource*(**this**,  R.array.***game\_status***, android.R.layout.***simple\_spinner\_item***); |

The first parameter is the context, the second parameter is the string-array (notice that we call it by using R.array) and the last parameter is what the layout of the Spinner should be.

We can also configure how the items should look like when the user clicks on the spinner:

|  |
| --- |
| *// Specify the layout to use when the list of choices appears*  statusAdapter.setDropDownViewResource(android.R.layout.***simple\_spinner\_dropdown\_item***); |

With this default layout, the items will appear as a popup in which you can select an item.

Now we have to set our adapter to the Spinner:

|  |
| --- |
| *//Set the adapter to the spinner*  **statusSpinner**.setAdapter(statusAdapter); |

Finally, we will set an OnClickListener on the save button, which will call the method to save the game, which we will make in a later substep:

|  |
| --- |
| **saveButton**.setOnClickListener(**new** View.OnClickListener() {  @Override  **public void** onClick(View v) {  saveGame();  }  }); |

**NOTE:** The saveGame method will be discussed in paragraph 4.5.

## 4.3: Creating helper methods:

We have to create some helper methods that method saveGame will use later. The first one is called getSimpleCurrentDate. When we create a new Game, the application will use the current date to set attribute ‘dateAdded’ of object Game. The problem is, Java’s default current Date objects are represented in date + time, and we only want the date(Note: I’m talking about Java.util.Date, not Java.sql.date). That why we’re going to use method getSimpleCurrentDate:

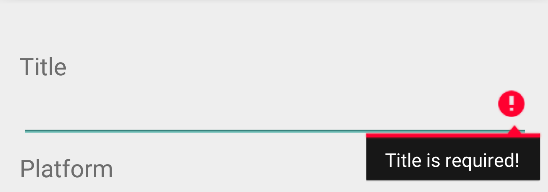
|  |
| --- |
| **private** Date getSimpleCurrentDate() {  Date curDate = **null**;  *//formatter that will convert dates into the day-month-year format*  SimpleDateFormat format = **new** SimpleDateFormat(**"dd/MM/yyyy"**);  *//Today's date, but with time included, which we don't want*  Date today = **new** Date();  **try** {  *//format.format returns a string, but we need a Date*  String curDateString = format.format(today);  *//Parse the date String into a Date object*  curDate = format.parse(curDateString);  } **catch** (ParseException ex) {  ex.printStackTrace();  }  **return** curDate;  } |

As you can see, we’re using a formatter to get the date into a numbered day-month-year. We get a String back when formatting, which we finally convert back to Date by parsing it.

The second helper method is called showToast. Create this method in AddGameActivity:

|  |
| --- |
| **private void** showToast(String message){  Context context = getApplicationContext();  **int** duration = Toast.***LENGTH\_SHORT***;  Toast toast = Toast.*makeText*(context, message, duration);  toast.show();  } |

The third method is to set error messages, and is called setErrorText. We are going to make the titleInput EditText and platformInput EditText mandatory, so we want to show error messages when they are not filled in. This helper method will take care of displaying a neat little error message. An example:



To make this happen, create method setErrorText:

|  |
| --- |
| **private void** setErrorText(EditText editText, String message){  *//get the color white in integer form*  **int** RGB = android.graphics.Color.*argb*(255,255,255,255);  *//Object that contains the color white*  ForegroundColorSpan fgcspan = **new** ForegroundColorSpan(RGB);  *//object that will hold the message, and makes it possible to change the color of the text*  SpannableStringBuilder ssbuilder = **new** SpannableStringBuilder(message);  *//give the message from the first till the last character a white color.*  *//The last '0' means that the message should not display additional behaviour*  ssbuilder.setSpan(fgcspan, 0, message.length(), 0);  *//Make the EditText display the error message*  editText.setError(ssbuilder);  } |

You may have noticed that an EditText and a message gets passed as parameters. This means that we can set messages for al EditText Views, such as EditText objects titleInput and platformInput. There is also a lot of tinkering with text color in this method. This is because not in all versions of android the text appears white. On some phones, the text color will appear black, exactly the same color as the background of the error message…

## 4.4: Creating class SharedPreferencesHelper

To save, retrieve, modify and delete our games inside SharedPreferences, we have to write a class that will take care of the communications with SharedPreferences. Create a new package called ‘utility’ and create a new Java class inside it, called SharedPreferencesHelper.

Add the following attributes and the constructor to the class:

|  |
| --- |
| *//name of the SharedPreferences file that we will use*  **private final** String **PREFERENCES\_FILE** = **"gameStorage"**;  *//name of the location where we've stored our games in SharedPreferences*  **private final** String **GAMES\_KEY** = **"games"**;  *//name of the location where we've stored the newest id*  **private final** String **ID\_KEY** = **"assignableId"**;  *//Object where we will store/retrieve games from*  **private** SharedPreferences **sharedPreferences**;  *//Object that will convert Objects from and to JSON*  **private** Gson **gson**;  *//ArrayList with our stored games*  **private** List<Game> **games** = **new** ArrayList<>(); |

Most of the comments speak for itself. SharedPreferences are files that contain data that can be used throughout multiple applications. We will use a SharedPreferences file called “gameStorage”, which will be created if it doesn’t exist yet when we save our first game.

The class also needs a constructor, where we instantiate our sharedPreferences and gson variables:

|  |
| --- |
| **public** SharedPreferencesHelper(Context context){  *//get the SharedPreferences from the context. If file gameStorage does not exist,*  *//it will be created when we commit our changes.*  *//MODE\_PRIVATE means that only this application can access file gameStorage*  **sharedPreferences** = context.getSharedPreferences(**PREFERENCES\_FILE**, Context.***MODE\_PRIVATE***);  **gson** = **new** Gson();  } |

Before we are going to make the save method, we need three helper methods. The first one is getGamesFromPreferences:

|  |
| --- |
| **private** List<Game> getGamesFromPreferences() {  *//Get the games list in JSON form from SharedPreferences on the location of GAMES\_KEY*  String gamesListJSON = **sharedPreferences**.getString(**GAMES\_KEY**, **"null"**);  **if** (!gamesListJSON.equals(**"null"**)) {  *//JSON was found in SharedPreferences*  *//Create a Type object that tells how we want our Games converted from JSON,*  *//we want an ArrayList with games in it*  *//Note: when importing, use 'java.lang.reflect.Type' and 'com.google.gson.reflect.TypeToken';*  Type type = **new** TypeToken<ArrayList<Game>>(){}.getType();  *//Get the ArrayList of games from the JSON String.*  List<Game> retrievedGames = **gson**.fromJson(gamesListJSON, type);  **return** retrievedGames;  } **else** {  *//No JSON was found in sharedPreferences, return an empty arrayList*  **return new** ArrayList<Game>();  }  } |

This method gets our list of games from SharedPreferences, if it exists, it will return an empty list otherwise. First we get the list of games in JSON format. Then we convert the JSON into an ArrayList of games. Then we use some tricky code to get the Type in which we want our games converted, which is ArrayList<Game>. Do pay attention when importing classes Type and TypeToken, there are multiple Type classes in Java. After getting the correct Type, we convert the JSON into an ArrayList of Games and return it.

The second method will save our list of games into SharedPreferences, called setGamesInPreferences:

|  |
| --- |
| **private void** setGamesInPreferences(List<Game> gamesToSave){  *//Get an editor to edit SharedPreferences*  SharedPreferences.Editor editor = **sharedPreferences**.edit();  *//Convert the list of games to a JSON string*  String gameListJSON = **gson**.toJson(gamesToSave);  *//Put the JSON inside SharedPreferences, assigned to a key (which is the value of GAMES\_KEY)*  editor.putString(**GAMES\_KEY**, gameListJSON);  *//Commit the changes*  editor.commit();  } |

To modify SharedPreferences, we need an Editor object. After doing that, we convert our list of games, that was passed as parameter, into a JSON string. We put this string inside SharedPreferences with the editor, and commit our changes.

The third helper method makes sure that the correct id gets assigned to a new game, and is called getAssignableId:

|  |
| --- |
| **private long** getAssignableId(){  *//Get the id that we can assign to a game*  **long** newId = **sharedPreferences**.getLong(**ID\_KEY**, 0);  *//Get an editor and update SharedPreferences with an ID that we can assign to a future game*  SharedPreferences.Editor editor = **sharedPreferences**.edit();  editor.putLong(**ID\_KEY**, newId + 1);  editor.commit();  **return** newId;  } |

The method gets the id from SharedPreferences, or returns 0 as default when it can’t find it. We then update the SharedPreferences by saving an incremented id to it, so that the next game will not get the same id as the game that we are trying to save right now.

We are now going to create the saveGame method.

|  |
| --- |
| **public void** saveGame(Game game) {  *//Get the most recent list of games from SharedPreferences*  **games** = getGamesFromPreferences();  *//Give the game the correct id, which we will get from SharedPreferences*  game.setId(getAssignableId());  **games**.add(game);  *//Save the updated gamelist in SharedPreferences*  setGamesInPreferences(**games**);  } |

As you can see, most of the stuff is done in the helper methods that we created.

## 4.5 Creating the saveGame method of AddGameActivity

At last, we can create the saveGame method. Open AddGameActivity, and create method saveGame:

|  |
| --- |
| **public void** saveGame(){  *//Get the current date in numbered day-month-year format*  Date curDate = getSimpleCurrentDate();  *//Retrieve the input from the user*  String title = **titleInput**.getText().toString();  String platform = **platformInput**.getText().toString();  String gameStatus = **statusSpinner**.getSelectedItem().toString();  String notes = **notesInput**.getText().toString();  **if**(title.equals(**""**)){  *//Make EditText titleInput display an error message, and display a toast*  *//that the title field is empty*  setErrorText(**titleInput**, getString(R.string.***title\_is\_required***));  showToast(getString(R.string.***title\_field\_is\_empty***));  } **else if** (platform.equals(**""**)){  *//Make EditText platformInput display an error message, and display a toast*  *//that the platform field is empty*  setErrorText(**platformInput**, getString(R.string.***platform\_is\_required***));  showToast(getString(R.string.***plaftorm\_field\_is\_empty***));  } **else** {  *//Create a SharedPreferencesHelper object, and pass it the context of this activity*  SharedPreferencesHelper preferencesHelper = **new** SharedPreferencesHelper(**this**);  *//Make a game object based on the input. The correct id will be set in preferenceHelper.saveGame()*  Game game = **new** Game(-1, title, platform, curDate, gameStatus, notes);  *//Save the game to sharedPreferences*  preferencesHelper.saveGame(game);  *//Notify the user with a toast that the game has been added*  showToast(getString(R.string.***game\_has\_been\_added***));  *//Go back to MainActivity*  Intent intent = **new** Intent(AddGameActivity.**this**, MainActivity.**class**);  startActivity(intent);  }  } |

We first get the current date, and the input from the user. Then we check if we need to display error messages if titleInput or platformInput is empty. If not, then we save the game, by passing it to our SharedPreferencesHelper object. After saving it, we return to MainActivity.

## 4.6: Adding navigation to addGameActivity

We still haven’t implemented navigation to AddGameActivity yet. Create‘menu\_main.xml’ (res > menu), and make it look like this:

|  |
| --- |
| <**menu xmlns:android="http://schemas.android.com/apk/res/android"**  **xmlns:app="http://schemas.android.com/apk/res-auto"**  **xmlns:tools="http://schemas.android.com/tools" tools:context=".MainActivity"**>  <**item android:id="@+id/action\_add\_game" android:title="@string/add\_game"**  **android:orderInCategory="1" app:showAsAction="ifRoom"** />  </**menu**> |

The **android:orderInCategory="1"** tells the order in which items need to appear. ‘1’ will appear first in the action bar, ‘2’ will appear after that, and so on.

**app:showAsAction="ifRoom"** Will make the button only visioble in the action bar if there is room. If not, it will move to the ‘three dots’ button in the action bar.

Open MainActivity, and add the following onCreateOptionsMenu. Make it look like this:

|  |
| --- |
| @Override  **public boolean** onCreateOptionsMenu(Menu menu) {  *// Inflate the menu; this adds items to the action bar if it is present.*  getMenuInflater().inflate(R.menu.***menu\_main***, menu);  **return true**;  } |

Open MainActivity, and add the onOptionsItemSelected. Make it look like this:

|  |
| --- |
| @Override  **public boolean** onOptionsItemSelected(MenuItem item) {  *// Handle action bar item clicks here. The action bar will*  *// automatically handle clicks on the Home/Up button, so long*  *// as you specify a parent activity in AndroidManifest.xml.*  **int** id = item.getItemId();  **if** (id == R.id.***action\_add\_game***){  Intent intent = **new** Intent(MainActivity.**this**, AddGameActivity.**class**);  startActivity(intent);  }  **return super**.onOptionsItemSelected(item);  } |

That’s it for the functionality to add a game. We’ll make the games appear in the list in MainActivity in the next step.

# **Step 5: Displaying our saved games**

We can save our games, but we cannot display them yet. Time to change this. First, we need to get our list of games from SharedPreferences. Open SharedPreferencesHelper.java, and add the following method:

|  |
| --- |
| **public** List<Game> getGames(){  *//return the ArrayList that we get from method getGamesFromPreferences*  **return** getGamesFromPreferences();  } |

After this, Open MainActivity, and go to method onCreate. Comment/delete all hardcoded games if you still have them. Make sure ListView gameList and (Array)List of Game objects is declared. And add this at the start of onCreate() to get the Games from the SharedPreferences.

|  |
| --- |
| *//Create a SharedPreferencesHelper object, and pass it the context of this activity*  SharedPreferencesHelper sharedPreferencesHelper = **new** SharedPreferencesHelper(**this**);  *//get the list of games from SharedPreferences*  **games** = sharedPreferencesHelper.getGames(); |

That’s it for displaying saved games, test the application to see if everything went well.

# **Step 6: Modifying games**

We also want to modify our games. Go to package activity and create a new Activity called ModifyGameActivity. We will navigate to ModifyGameActivity from GameDetailsActivity, so select GameDetailsActivity as hierarchical parent. A lot of code of the layout and the activity class looks like that of AddGameActivity, but with changes here and there.

## 6.1: Creating the layout

## 

## Create a layout so you can update all the different fields.

## 6.2: Initializing the Views and setting the listener



Initializing the Views is almost the same as AddGameActivity. The big difference is that we will set the data of the selected game into the fields. Open ModifyGameActivity and make sure all the data is set. You can use the following lines of code to get the data:

*//Get the selected game that we've sent from GameDetailsActivity*

Intent intent = getIntent();

**game** = (Game) intent.getSerializableExtra(**"currentGame"**);

There are a couple of things that we need to do, such as navigating from GameDetailsActivity to ModifyGameActivity + sending the selected game with it, creating helper methods, and creating modifyGame methods in both the activity and the SharedPreferencesHelper class. We will do these things in the next substeps.

## 6.3: Creating the helper methods

We also going to use a new helper method, called setSpinnerPosition:

|  |
| --- |
| **private void** setSpinnerPosition(ArrayAdapter adapter){  **if** (!**game**.getGameStatus().equals(**null**)){  *//Gets the position of the correct spinner item by comparing*  *//which item of the Spinner matches with the gameStatus*  **int** spinnerPosition = adapter.getPosition(**game**.getGameStatus());  *//Display the correct gameStatus in the Spinner based on the found position*  **statusSpinner**.setSelection(spinnerPosition);  }  } |

This method will make sure that status of the game is selected in the spinner when going to ModifyGameActivity by comparing the spinner items with the status of the game.

## 6.4: Adding method modifyGame to class SharedPreferencesHelper

We need to make a modifyGame method in SharedPreferencesHelper. We first need to make a small helper method for that, which we will also use for deleting games later. Open SharedPreferencesHelper.java, and add method getIndex:

|  |
| --- |
| **private int** getIndex(**long** id){  *//We will use int index to tell in which spot in the ArrayList we need to overwrite the old*  *//game with the new (modified) game*  **int** index = -1;  *//Determine index based on matching game id's*  **for** (Game tempGame : **games**){  **if** (tempGame.getId() == id){  *//Game to be modified found, retrieve its index*  index = **games**.indexOf(tempGame);  }  }  **return** index;  } |

Here, we compare the id’s of the outdated version of the game and the modified version of the game, so that we can get its index, and that we can replace the correct game in the gameslist.

Now we need to create method modifyGame (still in SharedPreferencesHelper):

|  |
| --- |
| **public void** modifyGame(Game game){  *//Get the most recent list of games from SharedPreferences*  **games** = getGamesFromPreferences();  *//get the index of the outdated game in the arraylist*  **int** index = getIndex(game.getId());  *//replace the old version of the game with the updated version*  **games**.set(index, game);  *//Save the updated gamelist in SharedPreferences*  setGamesInPreferences(**games**);  } |

Now we can create modifyGame in ModifyGameActivity.

## 6.5: Creating the modifyGame method in ModifyGameActivity

Open ModifyGameActivity and add method modifyGame:

|  |
| --- |
| **public void** modifyGame(){  *//Get the input from the Views*  String title = **titleInput**.getText().toString();  String platform = **platformInput**.getText().toString();  String gameStatus = **statusSpinner**.getSelectedItem().toString();  String notes = **notesInput**.getText().toString();  **if**(title.equals(**""**)){  *//Make EditText titleInput display an error message, and display a toast*  *//that the title field is empty*  setErrorText(**titleInput**, getString(R.string.***title\_is\_required***));  showToast(getString(R.string.***title\_field\_is\_empty***));  } **else if** (platform.equals(**""**)){  *//Make EditText platformInput display an error message, and display a toast*  *//that the platform field is empty*  setErrorText(**platformInput**, getString(R.string.***platform\_is\_required***));  showToast(getString(R.string.***plaftorm\_field\_is\_empty***));  } **else** {  *//update the game with the new data*  **game**.setTitle(title);  **game**.setPlatform(platform);  **game**.setGameStatus(gameStatus);  **game**.setNotes(notes);  *//Create a SharedPreferencesHelper object, and pass it the context of this activity*  SharedPreferencesHelper sharedPreferencesHelper = **new** SharedPreferencesHelper(**this**);  sharedPreferencesHelper.modifyGame(**game**);  *//Notify the user of the success*  showToast(getString(R.string.***game\_has\_been\_modified***));  *//Go back to ModifyGameActivity, and pass the updated game with is*  Intent intent = **new** Intent(ModifyGameActivity.**this**, GameDetailsActivity.**class**);  intent.putExtra(**"selectedGame"**, **game**);  startActivity(intent);  }  } |

As you can see, we get the input from the Views, check if the required fields are filled in, update the game with the new information, and save it to SharedPreferences. We then go back to GameDetailsActivity, where the game will be displayed with the new information.

## 6.6: Adding navigation to ModifyGameActivity

We only have to add navigation to the application to make the modification of games work. Create menu\_game\_details.xml (res > menu), and make it look like this:

|  |
| --- |
| <**menu xmlns:android="http://schemas.android.com/apk/res/android"**  **xmlns:app="http://schemas.android.com/apk/res-auto"**  **xmlns:tools="http://schemas.android.com/tools"**  **tools:context=".GameDetailsActivity"**>  <**item android:id="@+id/action\_modify\_game" android:title="@string/modify\_game"**  **android:orderInCategory="1" app:showAsAction="ifRoom"** />  </**menu**> |

Open GameDetailsActivity, and make method onCreateOptionsMenu look like this:

|  |
| --- |
| @Override  **public boolean** onCreateOptionsMenu(Menu menu) {  *// Inflate the menu; this adds items to the action bar if it is present.*  getMenuInflater().inflate(R.menu.***menu\_game\_details***, menu);  **return true**;  } |

Open GameDetailsActivity, and make method onOptionsSelected look like this:

|  |
| --- |
| @Override  **public boolean** onOptionsItemSelected(MenuItem item) {  **int** id = item.getItemId();  **if** (id == R.id.***action\_modify\_game***){  *//Go to ModifyGameActivity, and pass the current game with it to modify*  Intent intent = **new** Intent(GameDetailsActivity.**this**, ModifyGameActivity.**class**);  intent.putExtra(**"currentGame"**, **game**);  startActivity(intent);  }  **return super**.onOptionsItemSelected(item);  } |

If everything went correctly, you should now be able to modify games. Go ahead and test it.

# **Step 7: Deleting games**

So we can add, display and modify games. The only thing left to do is to delete a game. We are not going to make an activity for this, but a Dialog (mostly known as a popup) in the GameDetailsActivity.

## 7.1 Creating class ConfirmDeleteDialog

To create a Dialog, we must make a class for it. Go to package ‘utility’ and create a new Java class called ‘ConfirmDeleteDialog’. Make the class look like this:

|  |
| --- |
| **public class** ConfirmDeleteDialog **extends** DialogFragment {  */\* The activity that creates an instance of this dialog fragment must*  *\* implement this interface in order to receive event callbacks.*  *\* Each method passes the DialogFragment in case the host needs to query it. \*/*  **public interface** ConfirmDeleteDialogListener {  **public void** onDialogPositiveClick(DialogFragment dialog);  **public void** onDialogNegativeClick(DialogFragment dialog);  }  *// Use this instance of the interface to deliver action events*  ConfirmDeleteDialogListener **mListener**;  *// Override the Fragment.onAttach() method to instantiate the ConfirmDeleteDialogListener*  @Override  **public void** onAttach(Activity activity) {  **super**.onAttach(activity);  *// Verify that the host activity implements the callback interface*  **try** {  *// Instantiate the ConfirmDeleteDialogListener so we can send events to the host*  **mListener** = (ConfirmDeleteDialogListener) activity;  } **catch** (ClassCastException e) {  *// The activity doesn't implement the interface, throw exception*  **throw new** ClassCastException(activity.toString()  + **" must implement ConfirmDeleteDialogListener"**);  }  }  @Override  **public** Dialog onCreateDialog(Bundle savedInstanceState){  AlertDialog.Builder builder = **new** AlertDialog.Builder(getActivity());  builder.setMessage(R.string.***dialog\_game\_deletion\_confirmation***)  .setPositiveButton(R.string.***delete\_game***, **new** DialogInterface.OnClickListener() {  **public void** onClick(DialogInterface dialog, **int** id){  *//activate method onDialogPositiveClick inside implementing class*  **mListener**.onDialogPositiveClick(ConfirmDeleteDialog.**this**);  }  })  .setNegativeButton(R.string.***cancel***, **new** DialogInterface.OnClickListener(){  **public void** onClick(DialogInterface dialog, **int** id) {  *//activate method onDialogNegativeClick inside implementing class*  **mListener**.onDialogNegativeClick(ConfirmDeleteDialog.**this**);  }  });  **return** builder.create();  }  } |

The class holds an interface called ConfirmDeleteListener. Every class that wants to show this Dialog, needs to implement this interface. Method onAttach checks whether the interface was implemented in the class or not. Method onCreateDialog creates and shows the dialog. When the user clicks on the confirm or cancel button, the onDialogPositiveClick or onDialogNegativeClick methods inside the implementing class will be activated.

## 7.2: Add a delete button to the action bar

Open menu\_game\_details.xml and make it look like this:

|  |
| --- |
| <**menu xmlns:android="http://schemas.android.com/apk/res/android"**  **xmlns:app="http://schemas.android.com/apk/res-auto"**  **xmlns:tools="http://schemas.android.com/tools"**  **tools:context=".GameDetailsActivity"**>  <**item android:id="@+id/action\_modify\_game" android:title="@string/modify\_game"**  **android:orderInCategory="1" app:showAsAction="ifRoom"** />  <**item android:id="@+id/action\_delete\_game" android:title="@string/delete\_game"**  **android:orderInCategory="2" app:showAsAction="ifRoom"** />  </**menu**> |

## 7.3: Implementing the ConfirmDeleteDialog and link it to the delete button

Open GameDetailsActivity. Implement interface ConfirmDeleteDialog.ConfirmDeleteDialogListerner (this is the interface that resides inside class ConfirmDeleteDialog), so that the class header looks like this:

|  |
| --- |
| **public class** GameDetailsActivity **extends** AppCompatActivity **implements** ConfirmDeleteDialog.ConfirmDeleteDialogListener { |

It will give you an error message, because the required interface methods (onDialogPositiveClick and onDialogNegativeClick) are not implemented yet.

Time to do this:

|  |
| --- |
| @Override  **public void** onDialogPositiveClick(DialogFragment dialog) {  *//User clicked on the confirm button of the Dialog, delete the game from SharedPreferences*  SharedPreferencesHelper sharedPreferencesHelper = **new** SharedPreferencesHelper(GameDetailsActivity.**this**);  *//We only need the id of the game to delete it*  sharedPreferencesHelper.deleteGame(**game**.getId());  *//Game has been deleted, go back to MainActivity*  showGameDeletedToast();  Intent intent = **new** Intent(GameDetailsActivity.**this**, MainActivity.**class**);  startActivity(intent);  }  @Override  **public void** onDialogNegativeClick(DialogFragment dialog) {  *//Do nothing, Dialog will disappear*  } |

We don’t have a deleteGame method in SharedPreferencesHelper.java yet, we will make that in the next substep. The showGameDeletedToast method notifies the user that the game has been deleted, and looks like this:

|  |
| --- |
| **public void** showGameDeletedToast(){  Context context = getApplicationContext();  String text = getString(R.string.***game\_deleted***);  **int** duration = Toast.***LENGTH\_SHORT***;  Toast toast = Toast.*makeText*(context, text, duration);  toast.show();  } |

We still have to link the Dialog to the delete button in the action bar, otherwise the Dialog will never show up. Go to method onOptionsItemSelected (still in GameDetailsActivity), and make it look like this:

|  |
| --- |
| @Override  **public boolean** onOptionsItemSelected(MenuItem item) {  **int** id = item.getItemId();  **if** (id == R.id.***action\_modify\_game***){  *//Go to ModifyGameActivity, and pass the current game with it to modify*  Intent intent = **new** Intent(GameDetailsActivity.**this**, ModifyGameActivity.**class**);  intent.putExtra(**"currentGame"**, **game**);  startActivity(intent);  } **else if** (id == R.id.***action\_delete\_game***){  *//Show the ConfirmDeleteDialog*  DialogFragment dialog = **new** ConfirmDeleteDialog();  dialog.show(**this**.getFragmentManager(), **"ConfirmDeleteDialog"**);  }  **return super**.onOptionsItemSelected(item);  } |

Now the dialog will show up when we click on the delete button.

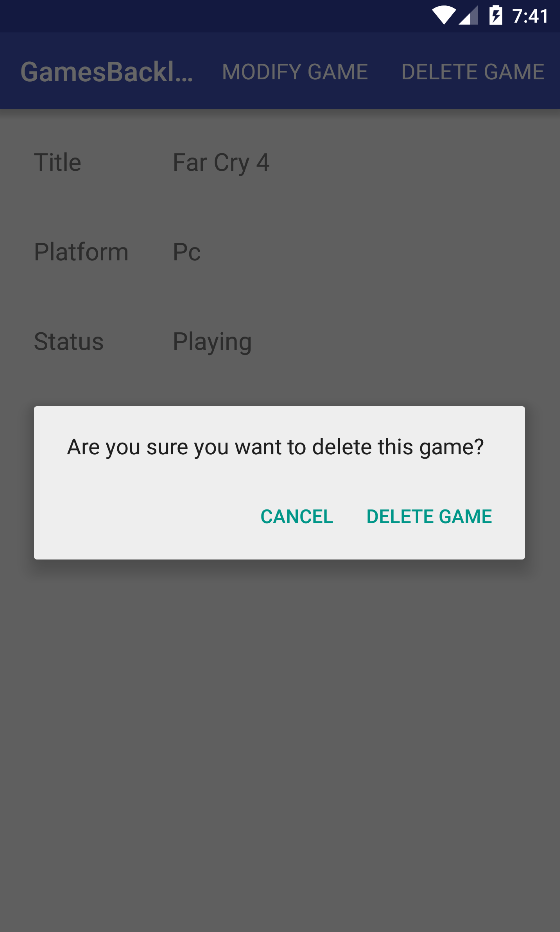
## 7.4: Creating method deleteGame in SharedPreferencesHelper

Open SharedPreferencesHelper.java, and add method deleteGame:

|  |
| --- |
| **public void** deleteGame(**long** id){  *//Get the most recent list of games from SharedPreferences*  **games** = getGamesFromPreferences();  *//get the index of the to be deleted game in the arraylist*  **int** index = getIndex(id);  *//Delete the game from the ArrayList, based on index*  **games**.remove(index);  *//Save the updated gamelist in SharedPreferences*  setGamesInPreferences(**games**);  } |

We first get the gamelist, then look up where the to be deleted game is positioned in the list, and then remove it. The list of games gets saved into SharedPreferences as final step.

This should be everything to delete a game.



And with that, the application is done!