POLYTECHNIC INSTITUTE OF LEIRIA

SCHOOL OF TECHNOLOGY AND MANAGEMENT

DEPARTMENT OF INFORMATICS ENGINEERING

MANAGING STATE AND DATA

The goal of this exercise sheet is to teach you to maintain activity state and persist (i.e., save and restore) important app data. You'll continue to work on the contacts app you started developing in the previous sheet. No new features will be added – but you'll make important modifications that will make your app a lot more usable.

Index

1) Singletons..2
2) Persisting App Data...4
3) Reading Data from a Local Text File..............7
4) Reading Data from a Remote Text File 10
5) Reading Data from a RESTful Web service...13

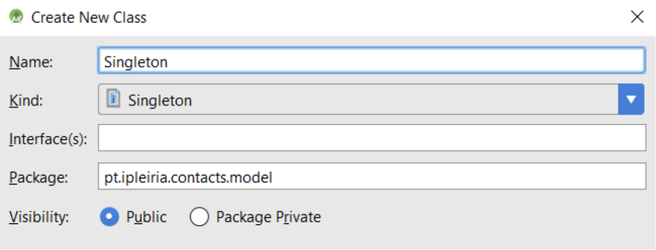
1) Singletons

Sometimes it's appropriate to have exactly one instance of a class – e.g., one instance of `Agenda`. Typically, those types of objects – known as *singletons* – are accessed by disparate objects throughout a software system (e.g., by several `Activity` objects), and therefore require a global point of access. Android's documentation[1] suggests utilizing Singletons for maintaining global application state.

Singletons maintain a static reference to the sole singleton instance, and return a reference to that instance from a static `getInstance()` method. With the Singleton design pattern you can:

- Ensure that only one instance of a class is created;
- Provide a global point of access to the object;
- Allow multiple instances in the future without affecting a singleton class's clients.

Create a new Singleton class in the model package: right-click on model; select New->Java class; name the class "Singleton"; and select "Singleton" as the class's "Kind" (Figure 1).



Create New Class

Name: Singleton

Kind: Singleton

Interface(s):

Package: pt.ipleiria.contacts.model

Visibility: ● Public ○ Package Private

Figure 1

Now edit the generated `Singleton` class in accordance to Code Block 1.

```
public class Singleton {
  private static final Singleton ourInstance = new Singleton();

  public static Singleton getInstance() {
    return ourInstance;
  }

  private Agenda agenda;

  private Singleton() {
    this.agenda = new Agenda();
  }

  public Agenda getAgenda() {
    return agenda;
  }

  public void setAgenda(Agenda agenda) {
    this.agenda = agenda;
  }
}
```

[1] https://developer.android.com/reference/android/app/Application.html

Code Block 1

From now on, for obtaining a reference to the `Agenda` object you'll be using the following instruction:

- `Singleton.getInstance().getAgenda()`

Now, the `MainActivity` no longer needs a field of type `Agenda`; the agenda will be stored in the Singleton. So, just delete the following instructions from your `MainActivity`:

- `private Agenda agenda;`
- `this.agenda = new Agenda();`

And replace all references to the (now deleted) `agenda` field with the abovementioned code snippet: `Singleton.getInstance().getAgenda()`

In fact, you no longer need to add the `Agenda` to the `Intent` as an extra when launching the `SearchActivity` and, as such, you can update your code accordingly: delete the call to `putExtra` from the `MainActivity`'s `onClick_SearchItem` method; and, in the `SearchActivity`, delete the instructions that obtains the `Agenda` from the `Intent` that started it.

Actually, you wouldn't even need have the new `Contact` returned from the `AddActivity` to the `MainActivity`; you could just add the new `Contact` to the agenda directly from the `AddActivity` using the Singleton. Try to implement this modification as an exercise.

More info: Simply Singleton – Java World, Singleton pattern - Wikipedia

2) Persisting App Data

In the previous section you learned how to maintain global application state, meaning that now your app is prepared to access the `Agenda` in whichever `Activity`. But what if you exit your application, and want your `Agenda` to be persisted (i.e., permanently saved) for when you run your Contacts App again – maybe even after you restart your phone?

For this you will need to use one of the available storage options provided by Android for you to save persistent application data: Shared Preferences (store private primitive data in key-value pairs); Internal Storage (store private data on the device memory); External Storage (store public data on the shared external storage); SQLite Databases (store structured data in a private database); Network Connection (store data on the web with your own network server).

Given that what we want is to persist our `Agenda`, we will be using *Internal Storage*, because: it's always available; files saved here are accessible by only your app by default; when the user uninstalls your app, the system removes all your app's files from internal storage.

You will be saving your data to the file system in the `onPause()` callback method (as suggested by Android's documentation) of the `MainActivity`, given that this is the last method that is sure to be called before the app's process is killed, and also because all the important data – namely, the `agenda` field – is contained in the `MainActivity`. So, let's edit `onPause()` in accordance to Code Block 2.

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

  try {
    FileOutputStream fileOutputStream =
        openFileOutput("agenda.bin", Context.MODE_PRIVATE);
    ObjectOutputStream objectOutputStream =
        new ObjectOutputStream(fileOutputStream);

    objectOutputStream.writeObject(Singleton.getInstance().getAgenda());

    objectOutputStream.close();
    fileOutputStream.close();
  } catch (IOException e) {
    e.printStackTrace();
    Toast.makeText(MainActivity.this,
        "Could not write Agenda to internal storage.",
        Toast.LENGTH_LONG).show();
  }
}
```

Code Block 2

When called, the `onPause()` method will now open the `agenda.bin` file (or create, if it does not already exist) in a private folder, create a stream for writing the `agenda` object, and write the `agenda`'s data to this file; finally, it will close both the stream and the object. If there is any problem, the `Exception` thrown will be printed in the log, and a `Toast` informing the user that something wrong happened will be shown.

It should be noted that only serializable objects (as are the `Agenda` and its `Contact`s) can be persisted to disk in this fashion.

Now we will also have to implement the code that is necessary for reading the agenda's data when the app is launched (or, more precisely, when the `MainActivity` is created). In order to do this, you will be modifying the `onCreate()` method as depicted in Code Block 3.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  try {
    FileInputStream fileInputStream = openFileInput("agenda.bin");
    ObjectInputStream objectInputStream = new
ObjectInputStream(fileInputStream);

    Singleton.getInstance().setAgenda(
        (Agenda) objectInputStream.readObject());

    objectInputStream.close();
    fileInputStream.close();
  } catch (FileNotFoundException e) {
    e.printStackTrace();
    Toast.makeText(MainActivity.this,
        "Could not read Agenda from internal storage (no Agenda yet?).",
        Toast.LENGTH_LONG).show();
  } catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
    Toast.makeText(MainActivity.this,
        "Error reading Agenda from internal storage.",
        Toast.LENGTH_LONG).show();
  }

  adapter = new ArrayAdapter<>(
      this,
      android.R.layout.simple_list_item_1,
      Singleton.getInstance().getAgenda().getContacts());
  listView = findViewById(R.id.listView_contacts);
  listView.setAdapter(adapter);
}
```

Code Block 3

Now the `onCreate()` method reads the data persisted to the internal storage basically by reverting the process described for recording the data to disk. First it tries to open the file, then it creates a stream reader, and finally it reads and reconstructs the serialized `agenda` and puts it in the `MainActivity` field. Finally, the file and the reader are closed.

If a problem happens an exception is thrown, and dealt with in the `catch` blocks. Notice that exceptions are now dealt with differently. Namely, if a `FileNotFoundException` is thrown, a different message is shown because this may not mean that there was an error – but simply that this is the first time that the application is run and that no data was saved yet.

So now just run your app, enter some data (you will have to enter all data by hand, given that now no dummy data is being inserted in the `onCreate()` method), exit you app (e.g., by pressing the back button on the `MainActivity` or by killing the app's process in the recently opened apps list), and launch it again. Is your agenda still there?

What if you want to erase all your Agenda data and start a fresh list of contacts? Can you find out how to do this?

More info: Saving Files, Storage Options, Saving Persistent State, Recreating an Activity

3) Reading Data from a Local Text File

In this section you are going to learn how to load some “dummy” contacts information from a text file that is stored locally.

Firstly, you need to create a text file and add the dummy data to it. Start by adding a *raw* resources folder: right-click your project, then click New->Android Resource Directory. Next, add a text file called `contacts.txt` to it: right-click your raw folder, then click New->File.

Files that you save in the raw folder are compiled into an `.apk` file as-is. Since raw is a subfolder of Resources (*res*), Android will automatically generate an ID for any file located inside it. This ID is then stored at the `R` class that will act as a reference to a file, meaning it can be easily accessed from other Android classes and methods and even in Android XML files.

More info: [Android Application Modules](), [Providing Resources]()

Now, copy paste the contents of Code Block 4 to your `contacts.txt` file, and save it.

```
# id:name
961223344:Francisco Stromp
932323232:Fernando Peyroteo
915566677:Vítor Damas
967659876:Rui Jordão
234678905:Joseph Szabo
234354565:Krasimir Balakov
922635106:Manuel Fernandes
961942690:Hector Yazalde
```

Code Block 4

Proceed to add a new a new menu item to the `menu_main.xml` file (Code Block 5), and add the necessary code for handling the click in the newly created “Read Contacts from Text File” menu item (Code Block 6).

```
<item android:id="@+id/action_read_contacts_text_file"
      android:title="Read Contacts from Text File"
      android:onClick="onClick_read_contacts_text_file"
      app:showAsAction="ifRoom" />
```

Code Block 5

```
public void onClick_read_contacts_text_file(MenuItem item) {
  try {
    InputStream inputStream =
        getResources().openRawResource(R.raw.contacts);

    String contentAsString = readStream(inputStream);
    inputStream.close();

    parseContacts(contentAsString);

    adapter.notifyDataSetChanged();

    Toast.makeText(MainActivity.this,
        "Contacts loaded from file!", Toast.LENGTH_LONG).show();
  } catch (IOException e) {
    Toast.makeText(MainActivity.this,
        "Error reading text file...", Toast.LENGTH_LONG).show();
    e.printStackTrace();
  }
}
```

Code Block 6

When you click on the "Read Contacts from Text File" menu item, your text file will be opened and an `InputStream` will be created that allows the file's contents to be loaded to a `String` variable; for this purpose, the `readStream()` method (to be implemented) is called. After the `InputStream` is closed (because you've already loaded all the file's contents anyway) the string is parsed using the `parseContacts()` method (to be implemented). Finally, you send notice that the data has changed to the `ListView` (via the adapter), and present a `Toast` informing the user that the contacts were loaded successfully; if anything goes wrong and an `IOException` is thrown, the user is also informed, and the stack trace is sent to the log allowing developers to trace the origin of the problem if needed.

Finally, you must append the `readStream()` (Code Block 7) and the `parseContacts()` (Code Block 8) methods to the `MainActivity` class.

```
private String readStream(InputStream is) {
  StringBuilder sb = new StringBuilder(512);
  try {
    Reader r = new InputStreamReader(is, "UTF-8");
    int c = 0;
    while ((c = r.read()) != -1) {
      sb.append((char) c);
    }
  } catch (IOException e) {
    throw new RuntimeException(e);
  }
  return sb.toString();
}
```

Code Block 7

```
private void parseContacts(String text) {
  String[] lines = text.split("\n");

  for (String line : lines) {
    if (line != null && !line.startsWith("#") && !line.trim().isEmpty()) {
      String[] split = line.split(":");

      int contactPhone = Integer.parseInt(split[0]);
      String contactName = split[1].trim();

      try {
        Contact contact = new Contact(contactName, contactPhone);
        Singleton.getInstance().getAgenda().addContact(contact);
      } catch (RuntimeException e) { // phone already exists
        Toast.makeText(this,
            e.getMessage(), Toast.LENGTH_SHORT).show();
        e.printStackTrace();
      }
    }
  }
}
```

Code Block 8

The `readStream()` method is a general-purpose method for loading the contents of a text file onto a String (which is returned).

The `parseContacts()` method, on the other hand, is prepared to parse information with the pattern described in the first line of the `dummy_contacts.txt` file: all the contacts' information (names and ids) are obtained, and new `Contact`s are created and added to the `Agenda` field of the `MainActivity`.

Now, try to analyse the `parseContacts()` method's code. Is it possible to have more than one contact per line? And how are empty lines dealt with? Is it possible for a line to contain both contact information and a comment?

4) Reading Data from a Remote Text File

In the previous section you learned how to read data from a text file. But what if the data is stored on a remote server? This section shows you how to connect to the network, and explains some of the best practices you should follow in creating even the simplest network-connected app.

Firstly, to perform the network operations described in this section, your application must include permission for accessing the internet and the network's state. Paste the markup available in Code Block 9 to your manifest file (inside the `<manifest>` element).

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.INTERNET"/>
```

Code Block 9

More info: App permissions

Now proceed to add a new a new menu item to the `menu_main.xml` file (Code Block 10), and then add the necessary code for handling the click in the newly created "Read Contacts from Network" menu item (Code Block 11) to the `onOptionsItemSelected()` method of the `MainActivity` class.

```
<item android:id="@+id/action_read_contacts_text_file"
      android:title="Read Contacts from Remote Text File"
      android:onClick="onClick_read_remote_text_file"
      app:showAsAction="ifRoom" />
```

Code Block 10

```
public void onClick_read_remote_text_file(MenuItem item) {
  ConnectivityManager connMgr =
     (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
  NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();

  if (networkInfo != null && networkInfo.isConnected()) {
    DownloadContactsTask downloadContactsTask =
                                               new DownloadContactsTask();
    downloadContactsTask.execute(URL_REMOTE_TEXT_FILE);
  } else {
    Toast.makeText(MainActivity.this,
               "Error: no network connection.", Toast.LENGTH_LONG).show();
  }
}
```

Code Block 11

Let's now go through the code for handling the download. Before your app attempts to connect to the network, it should check to see whether a network connection is available because the device may be out of range of a network or the user may have disabled both Wi-Fi and mobile data access. This can be accomplished by obtaining a `NetworkInfo` object using the `getActiveNetworkInfo()`, and then checking if the `isConnected()` method returns true.

If there is, in fact, a connection, the app proceeds to instantiate a `DownloadContactsTask` (to be implemented), and calling that `execute()` method on it. This method receives a `String` containing the URL of the remote text file, and proceeds to download it.

For the purpose of this example, the text file described in Code Block 4 was uploaded to the cloud using a free service available at https://goo.gl/ZKF9TA. Use it to upload your own text file, and store its URL in the `URL_REMOTE_TEXT_FILE` constant of the `MainActivity` class.

Let's now implement the `DownloadContactsTask` class as an inner class of the `MainActivity` class. In order to do this, copy the code depicted in Code Block 12 to your `MainActivity` class.

```java
private class DownloadContactsTask extends AsyncTask<String, Void, String>
{
  @Override
  protected String doInBackground(String... urls) {

    try {
      // establish the connection to the network resource
      URL url = new URL(urls[0]);
      HttpURLConnection httpURLConnection =
          (HttpURLConnection) url.openConnection();
      httpURLConnection.setReadTimeout(10000);
      httpURLConnection.setConnectTimeout(15000);
      httpURLConnection.setRequestMethod("GET");
      httpURLConnection.setDoInput(true);
      httpURLConnection.connect();

      int responseCode = httpURLConnection.getResponseCode();
      Log.i("Contacts App", "HTTP response code: " + responseCode);

      //retrieve the network resource's content
      InputStream inputStream = httpURLConnection.getInputStream();
      String contentAsString = readStream(inputStream);
      inputStream.close();

      return contentAsString;
    } catch (IOException e) {
      return "ERROR: unable to retrieve web page. URL may be invalid.";
    }
  }

  @Override
  protected void onPostExecute(String result) {
    if (!result.startsWith("ERROR")) {
      parseContacts(result);

      adapter.notifyDataSetChanged();

      Toast.makeText(MainActivity.this,
          "Contacts loaded from remote file!", Toast.LENGTH_LONG).show();
    } else {
      Toast.makeText(MainActivity.this, result, Toast.LENGTH_LONG).show();
    }
  }
}
```

Code Block 12

Before discussing the DownloadContactsTask's code, you must first comprehend that network operations can involve unpredictable delays, and that to prevent this from causing a poor user experience you should always perform network operations on a separate thread from the User Interface (UI). The AsyncTask class provides one of the simplest ways to fire off a new task from the UI thread; it allows to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers.

More info: AsyncTask, Multithreading for Performance

AsyncTask must be subclassed to be used. The subclass will override at least one method (doInBackground(Params...)), and most often will override a second one (onPostExecute(Result)) – as is the case with the DownloadContactsTask class.

The three generic types used by an AsyncTask<Params, Progress, Result> are the following:

- Params, the type of the parameters sent to the task upon execution;
- Progress, the type of the progress units published during the background computation; and
- Result, the type of the result of the background computation.

Not all types are always used by an asynchronous task. To mark a type as unused, simply use the type Void (as with the Progress type in the DownloadContactsTask, for example).

The DownloadContactsTask implements the following AsyncTask methods:

- doInBackground() receives the text file's URL as a parameter, establishes the connection to the network resource, and retrieve the network resource's content (using the readStream() method discussed in the previous Section). When it finishes, it passes back a result string containing the remote text file's contents.
- onPostExecute() takes the returned string, parses it (using the parseContacts() method discussed in the previous Section), and displays the downloaded contacts in the UI's List View.

The connection and retrieval of the network resource's content, in particular, is performed using HttpURLConnection client, which uses HTTP to send and receive data over the web, and supports HTTPS, streaming uploads and downloads, configurable timeouts, IPv6, and connection pooling.

Note that the method getResponseCode() returns the connection's status code. This is a useful way of getting additional information about the connection. A status code of 200 indicates success.

More info: HttpURLConnection, Connecting to the Network

5) Reading Data from a RESTful Web service

A web service is a standard for exchanging information between different types of applications irrespective of language and platform. This section shows you how to consume information from a RESTful web service.

You will be employing the Volley and Gson libraries:

- *Volley* is an HTTP library that makes networking for Android apps easier and faster. It provides built-in support for raw strings, images, and JSON, among many other feature, allowing you to concentrate on the logic that is specific to your app.
- *Gson* is a Java library that can be used to convert Java Objects into their JSON representation, or vice-versa. It can work with arbitrary Java objects – including pre-existing objects.

More info: Transmitting Network Data Using Volley, Gson User Guide

You will call a Web Service operation named `GetContacts` that returns a list of contacts stored in a remote server in JSON format (or, optionally, XML). For accessing this operation, the following URL will be used:

- http://contactswebservice.apphb.com/Service.svc/rest/contacts

Because it is an HTTP GET operation, you can try it out simply by copy-pasting it to your browser's address bar; it should return the list of contacts currently stored in the Web Service (in XML, given that this is the default response format). However, to have more control over invocation, you can use test clients such as the *Advanced REST Client*[2] or *Postman*[3]. If you use one of these test clients, you will be able to define the "Content-Type" as "application/json" in the *header* of your *request*; this way, the Web Service's response should be similar to that depicted in Code Block 13.

```
[
  {
    "Name": "Justino Figueiredo",
    "Phone": 964574277
  },
  {
    "Name": "Emilia Figueiredo",
    "Phone": 919345876
  }
]
```

Code Block 13

JSON (JavaScript Object Notation) is a lightweight data-interchange format; which is easy for humans to read and write, and for machines to parse and generate. It is a *de facto* standard for storing and transmitting data in modern web application. The sample JSON response depicted in Code Block 13 contains an *array* – which begins with [(left bracket), ends with] (right bracket), and contains values are separated by , (comma). This array

[2] https://install.advancedrestclient.com/

[3] https://www.getpostman.com/

contains two objects. An *object* is an unordered set of name/value pairs – which begins with { (left brace) and ends with } (right brace); each name is followed by : (colon) and the name/value pairs are separated by , (comma).

More info: JSON

The help page of the `GetContacts` operation describing the format of the JSON response is available here:

- http://contactswebservice.apphb.com/Service.svc/rest/help/operations/GetContacts

Let's then proceed to implement this functionality in your app. You'll have to: invoke the `GetContacts` operation in the Web Service using the Volley library; parse the response received using Gson library; and update the Agenda with the new contacts received.

Remember that to perform the network operations described in this section your application must include the permission depicted in Code Block 9. In order to use Volley and Gson with Gradle/Android you will also have to add the necessary dependencies to the "build.gradle (Module: app)" file, as shown in Code Block 14.

```
dependencies {
    (…)

    compile 'com.android.volley:volley:1.1.0'
    compile 'com.google.code.gson:gson:2.8.2'
}
```

Code Block 14

Now proceed to add a new item to your menu, so as to enable the implementation of this functionality (Code Block 15).

```
<item android:id="@+id/action_network"
      android:title="Get Contacts from Web Service"
      android:onClick="onClick_NetworkItem"
      app:showAsAction="ifRoom" />
```

Code Block 15

Proceed to include the method for handling the click in the menu item in your `MainActivity` file (Code Block 16).

```
public void onClick_NetworkItem(MenuItem item) {
  RequestQueue queue = Volley.newRequestQueue(this);

  final String url =
      "http://contactswebservice.apphb.com/Service.svc/rest/contacts";

  JsonArrayRequest request = new JsonArrayRequest(
      Request.Method.GET,
      url,
      null,

      new Response.Listener<JSONArray>() {
        @Override
        public void onResponse(JSONArray response) {
          Log.d("Response", response.toString());

          Gson gson = new Gson();
          Contact[] contacts = gson.fromJson(
              response.toString(), Contact[].class);

          for (Contact contact : contacts) {
            Singleton.getInstance().getAgenda().addContact(contact);
          }

          adapter.notifyDataSetChanged();
        }
      },

      new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {
          Log.e("Response", error.getMessage());

          Toast.makeText(MainActivity.this,
              "Error retrieving contacts from webservice.",
              Toast.LENGTH_SHORT).show();
        }
      }
  )
  {
    @Override
    public Map<String, String> getHeaders() {
      HashMap<String, String> headers = new HashMap<>();
      headers.put("Content-Type", "application/json");
      return headers;
    }
  };

  queue.add(request);
}
```

Code Block 16

To call the Web Service operation, you start by obtaining a new `RequestQueue` object – which will contain the Network/HTTP requests that need to be made. Calling `add(Request)` will enqueue the given `Request` for dispatch – which is exactly what you'll be doing once you're finished creating the `Request`.

Next, you set the `url` variable as being the address of the abovementioned `GetContacts` operation.

The following step is indeed to create a `Request` – more precisely a `JsonArrayRequest` given that you want to retrieve a `JSONArray` response body (such as the one in Code Block 13). The HTTP method to use is GET; for the `url` we use the variable defined earlier; we set the `jsonRequest` as *null*, which indicates no parameters will be posted along with request; and we finally proceed to define the listener to receive the JSON response, and the error listener.

The listener is the callback object for delivering parsed responses; this means that once a response is available, the `onResponse` method will be called. In this method you: instantiate a new `GSon` object to help obtaining the `Contact`s from the JSON response; use the `fromJson` method to deserialize the Json string into an object of the specified class (which, in this case, is in fact an array of `Contacts`); add the contacts to the Agenda; and refresh the UI.

The `onErrorResponse` method is called if an error has occurred (with the provided error code and optional user-readable message). We simply log it and provide the user with an indication that something went wrong.

We also override the `getHeaders` method to have it return an extra HTTP header to go along with this request, which defines “Content-Type” as "application/json" (otherwise, the response format will be XML).

More info: Android Volley Tutorial – Making HTTP GET, POST, PUT, and DELETE Requests

Finally, you’ll have you make some adaptations to the `Contact` class – because the properties of the JSON objects retrieved from the Web Service don’t map directly to the `Contact`’s attribute names. If you look closely, the names of the JSON objects’ properties start with a capital letter (“Name” and “Phone”), whereas the attributes of the Contact class start with lowercase letter (“name” and “phone”).

In order to fix this, you’ll have to employ the `@SerializedName` annotation, as in Code Block 17.

```
public class Contact implements Serializable {
  @SerializedName("Name")
  private String name;

  @SerializedName("Phone")
  private int phone;

   (…)
}
```

Code Block 17

More info: JSON Field Naming Support (Gson)

That’s it! Go ahead and connect to the cloud.