Lappeenrannan teknillinen yliopisto

School of Business and Management

Sofware Development Skills

<Daria Zakharova>, <Insert student number here>

LEARNING DIARY, <MOBILE> MODULE

**LEARNING DIARY**

**21.09.2025**

I installed Android Studio and JDK 17 (Java SDK) and verified the setup with java -version / javac -version.

I created my first Android project TestApplication using Empty Views Activity (Java, XML layouts; Gradle with version catalog). Built a tiny calculator UI in activity\_main.xml (two EditTexts, one Button, one TextView) using ConstraintLayout. Implemented click handling in MainActivity.java: read numbers from the two inputs, add them, and show the result in the TextView.

I ran the app from Android Studio and tested the flow end-to-end.

Also I set up Git: created repo structure (lut-sds-mobile-2025/ with TestApplication/ + Coursework/), added an Android .gitignore, and made my initial commits.

I learned Android project structure: 1) MainActivity.java controls logic and res/layout/activity\_main.xml defines the UI; 2) ConstraintLayout basics: every view needs both horizontal and vertical constraints. Design-time positions aren’t used at runtime; 3) event handling; 4) debugging: running with the debugger, setting breakpoints, and inspecting variables in Android Studio.

Time spent: ~3 hours (setup, project creation, coding, debugging, repo setup).

**6.10.2025**  
  
I watched the second “Core elements” tutorial and deepened my understanding of Android basics. I reviewed what an Activity is (the top-level screen the user sees), how onCreate() runs first when an Activity loads, and how findViewById(...) lets me reference views from code. I also learned the Intent model: an Intent represents an action to perform; I practiced startActivity(...) to launch screens, putExtra(...) to attach data, and reading it back with getIntent().getStringExtra(...).

I initialized a new app (Empty Views Activity, Java) and created a SecondActivity. From MainActivity I wired a button to open SecondActivity via an explicit Intent and passed a string as a key-value pair, which I then displayed in the second screen as a TextView. In addition to in-app navigation, I tried an implicit Intent to leave my app: a “GOOGLE” button opens the device browser to https://www.google.com. I added CATEGORY\_BROWSABLE and wrapped startActivity(...) in a try/catch (ActivityNotFoundException) to handle cases where no browser is available (this wasn’t in the tutorial, I found it online because my Android version is much newer than in the video).

**15.10.2025**

I continued with the Android development tutorial (Part 3) and watched just a half of it. The focus was on learning to display lists using the *ListView* component. I followed the instructor to build a small “List App” that shows products with their name, price, and description.

First, I defined the data in *res/values/strings.xml* using three parallel *string-arrays* (items, prices, descriptions). Then, in the layout *activity\_main.xml*, I added a *ListView* underneath a toolbar inside a *ConstraintLayout*.

In *MainActivity.java* I loaded the arrays from resources and created a custom adapter (*ItemAdapter*) to connect the data with the *ListView*. I implemented the adapter by extending *BaseAdapter*, inflating a custom row layout (*my\_listview\_detail.xml*), and binding each item’s name, description, and price to *TextView* elements.

At first the app crashed because the *getView()* method returned null. I fixed it by returning the inflated view. After that the app ran, but the list appeared empty - caused by incorrect layout parameters (each row was taking the full screen). Adjusting the row layout to *wrap\_content* heights made the list display correctly.

I learned:

* How *ListView* works with a custom adapter and layout inflater.
* How Android resource arrays can provide structured data for lists.
* How returning the proper view in *getView()* and defining reasonable layout constraints are essential to rendering UI elements correctly.

**16.10.2025**

I continued with the second half of the Part 3 tutorial, where we extended the previous “List App” by adding image display functionality using *ImageView* and a second screen (*DetailActivity*).

In *MainActivity.java*, I implemented an *OnItemClickListener* for the *ListView* so that when a user taps an item, an *Intent* launches *DetailActivity* and passes along the item’s index using *putExtra()*.

In the new *DetailActivity.java*, I used that index to select the corresponding image resource and display it inside an *ImageView*. I created a helper method *getImg(int index)* to map each item to its drawable (peach, tomato, squash). Another method, *scaleImg(),* handled image resizing so that large images fit the screen properly. This was done using the *BitmapFactory* class to decode the image efficiently and scale it relative to the device’s screen width.

Also a new layout file, *activity\_detail.xml*, which contains an *ImageView*, was created. Finally, I added three sample images to the res/drawable folder and tested navigation between the list and detail screens - tapping an item now opens its picture on a separate page.

I learned:

* How to handle item click events in a *ListView* and navigate between activities using *Intent* and extras.
* How to display images dynamically with *ImageView* and *BitmapFactory*.
* How to decode and scale bitmaps efficiently to match screen dimensions.
* How to structure multi-activity apps with distinct layouts and data flow between them.

**26.10.2025**

I started working on my final project for the Mobile module - a simple **Expense Tracker app**. After reviewing the project requirements, I decided to build an app that records daily expenses, displays them in a list, and calculates the total amount spent.

I initialized a new Android Studio project named *ExpenseTracker* (Empty Views Activity, Java) and configured the base app theme and toolbar. I then created the main view structure (*activity\_main.xml*) with a *MaterialToolbar* at the top, a *TextView* showing the total amount, and a *ListView* below it to display expense items. I customized layout constraints and margins to achieve a clean look and proper spacing.

Next, I defined a simple data model *Expense.java* to represent an expense with two properties: *name* and *amount*. To display these objects in the list, I created a custom adapter *ExpenseAdapter.java*. Inside it, I implemented methods to inflate a custom layout (*row\_expense.xml*) for each row and bind the data to two *TextViews* (expense name and amount).

Finally, I added mock data directly in *MainActivity.java* to test the setup - three sample items. I verified that the data appeared correctly in the *ListView* and that the layout behaved responsively with proper spacing and text formatting.

**26.10.2025**

I continued developing my *Expense Tracker app* by expanding its functionality to allow users to add new expenses through a separate screen. First, I designed and integrated an “Add Expense” button on the main screen. After researching suitable Material Design components, I found the *Floating Action Button* to be the most appropriate for this purpose, as it visually represents a primary action. I added it to the bottom-right corner of the main layout and connected it to open a new *AddExpenseActivity* when tapped.

Next, I created the *Add Expense layout*, which includes two input fields (name and amount) and a “Save” button. I also added a back arrow in the top toolbar that allows users to return to the main screen without saving.

After building the UI, I implemented the logic to pass data from *AddExpenseActivity* back to *MainActivity* using the modern *ActivityResultLauncher* API. When a user enters valid data and clicks “Save,” the activity returns the expense details to the main screen, where the new entry is added to the list and the total sum updates automatically. I debugged several issues (such as incorrect button IDs) to make sure the data flow works correctly.

I learned:

* How to find and apply the right Material Design component (FloatingActionButton) by consulting official documentation.
* How to create and wire a multi-screen form for user input.
* How to handle navigation using toolbar back icons.
* How to implement the *ActivityResultLauncher* pattern to return data between Activities.

**27.10.2025**

Today I extended my *Expense Tracker app* with category functionality and began preparing a new *Summary page*.

First, I enhanced the *Add Expense view* by adding a *Spinner* for selecting a category (for example, Food, Transport, Shopping, Other). The *Spinner* provides a dropdown list of predefined categories, and if the user doesn’t select any, the category defaults to “Other.” This required updating the layout (*activity\_add\_expense.xml*) as well as extending the *Expense class* and *ExpenseAdapter* so that each expense item now displays its category alongside its name and amount on the main screen.

After that, I added a “Show Summary” button next to the total amount on the main page. Clicking this button now opens a new *SummaryActivity*, which will later display statistical information about the user’s spending by category. At this stage, the *Summary page* contains only a toolbar and placeholder text, but navigation between screens already works correctly.

Through these updates, I practiced:

* How to use a *Spinner* to offer selectable options in a form and how to handle its default value.
* How to implement multi-screen navigation by adding a *new Activity* and connecting it via a button and an *Intent*.

29.10.2025

I continued developing the *Expense Tracker app* by adding new functionality for both data visualization and interactivity.

First, I implemented a *Summary page*, accessible from the “Show Summary” button on the main screen. This new screen receives categorized expense data (category names, totals, and overall sum) via *Intent extras* from the *MainActivity*. I learned to aggregate data in Java using *HashMap* and pass structured lists (*ArrayList<String>, ArrayList<Float>*) between activities. On the *SummaryActivity* side, I built a layout that displays per-category totals and their percentage share of the overall spending.

Next, I added delete functionality to the expense list. I implemented a custom callback interface *OnExpenseDeleteListener* inside *ExpenseAdapter*, which notifies the *MainActivity* when a deletion occurs. The adapter is then refreshed with *notifyDataSetChanged(),* and the *updateTotal()* method recalculates the displayed total dynamically.

I learned:

* How to compute grouped statistics in Java and pass them safely between activities using *Intents*.
* How to use a callback interface to communicate user actions (like deleting a list item) from an adapter to an activity.