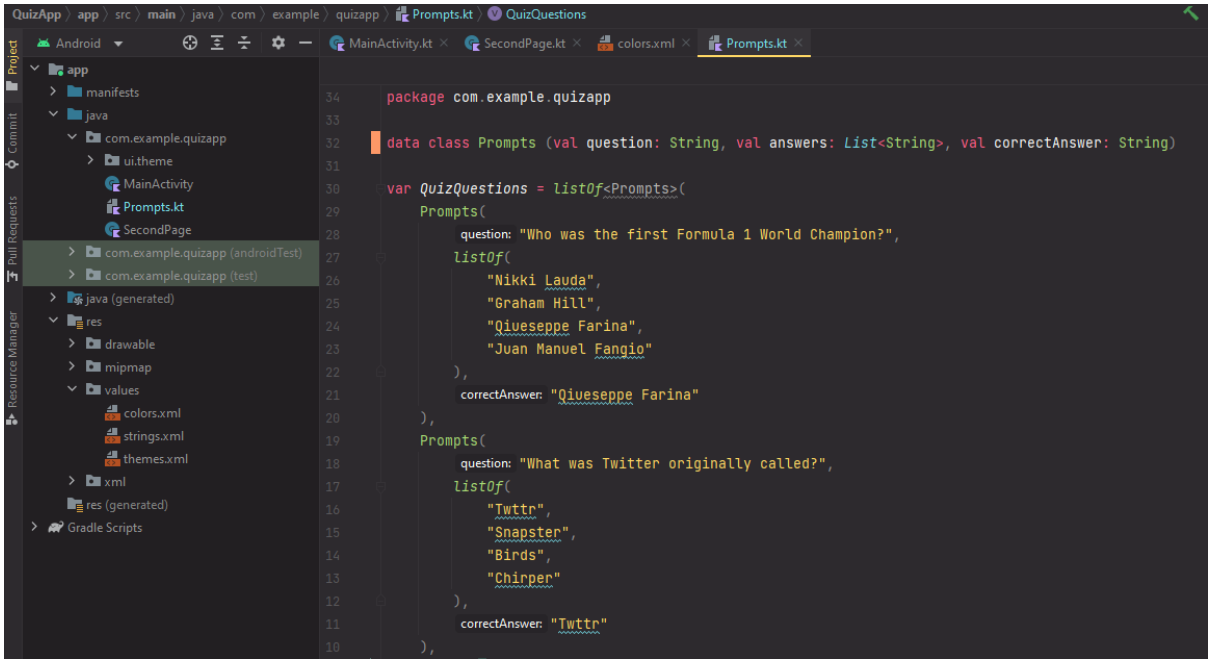
Compose UI

Saku Karttunen – 1.12.2022

Tein monen tehtävän Composella, sillä pidän Composen tuomasta tavasta kirjoittaa sovelluksia, sillä se muistuttaa enemmän minua web ohjelmoinnista, josta minulla on jo jonkin verran kokemusta. Compose myös vähentää tarvittavan koodin määrää, joka on erittäin hyvä bonus.

Valitsin esiteltäväksi tehtävän 3 – Quiz App. Sillä se oli mukava rakentaa ja pidin prosessista paljon.



Tein tehtävää varten oman tiedoston, johon rakensin data class:in kysymyksiin liittyvää dataa varten, jotta kysymyksien dataan olisi helppo päästä käsiksi. Tämän lisäksi kaikki kysymykset saatiin yhteen listaan todella helposti. Tämä helpotti ja nopeutti ohjelmointia funktioiden kohdalla.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // Shuffle the order of the questions at the creation
        // no need to shuffle them more than once per game
        QuizQuestions = QuizQuestions.shuffled()
        setContent {
            QuizAppTheme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    // Background color
                    color = colorResource(id = R.color.darkGray),
                ) {
                    MainCompose()
                }
            }
        }
    }
}
```

Main activityssä rakennusvaiheessa, käytin Kotlinin sisäänrakennettua .shuffled() funktiota. Tämä varmisti sen, että kysymykset tulevat aina satunnaisessa järjestyksessä. Se on heti ensimmäisenä, sillä sen tarvitsee vain shufflata kysymykset kerran heti sovellusta avatessa.

```
14
15      @Composable
16      fun MainCompose() {
17          Surface(
18              modifier = Modifier
19                  .fillMaxSize()
20                  .padding(vertical = 40.dp, horizontal = 20.dp),
21              // Foreground color
22              contentColor = colorResource(id = R.color.mutedWhite),
23              // Background color
24              color = colorResource(id = R.color.darkGray),
25          ) {
26              Column (
27                  horizontalAlignment = Alignment.CenterHorizontally
28              ) { this: ColumnScope
29                  MainTitle(name = "Quiz App")
30              }
31              QuizPrompt()
32          }
33      }
```

Composablet oli helppo myös jakaa pienempiin osiin, jotta niitä oli helppo sitten sijoitella näytölle sopivalla tavalla.

```
53
54  @OptIn(ExperimentalFoundationApi::class)
55  @Composable
56  fun QuizPrompt() {
57      // Initialize the variables, so that the UI will update when they change
58      var promptCounter by remember { mutableStateOf( value: 0) }
59      var question by remember { mutableStateOf( value: "") }
60      val userScore = remember { mutableStateListOf<String>() }
61      val correctAnswers = mutableListOf<String>()
62
63      // get correct answers
64      // for loop has the same kind of syntax as in javascript
65      // except of = in
66      for (prompt: Prompts in QuizQuestions) {
67          correctAnswers.add(prompt.correctAnswer)
68      }
```

Itse kysymysten compose elementissä, aluksi tein kaikista tarvittavista muuttujista remember muotoisia, jotta niiden muutokset aiheuttaisivat uudelleen piirtämisen. UserScore on myös remember muodossa, sillä käytin sitä debuggauksessa teksi elementissä.

Kysymykset myös muokataan ja siirretään omaan listaansa, jotta myöhemmin niihin on helppo verrata.

```
69
70      // get the current prompt
71      val currentPrompt = QuizQuestions[promptCounter]
72      // initialize the question
73      question = currentPrompt.question
74
75      // var answers = remember { mutableStateListOf<String>() }
76      var answers = mutableListOf<String>()
77      answers.add(currentPrompt.answers[0])
78      answers.add(currentPrompt.answers[1])
79      answers.add(currentPrompt.answers[2])
80      answers.add(currentPrompt.answers[3])
81
82      // shuffle answers, so they aren't in the same place every time
83      answers = answers.shuffled() as MutableList<String>
```

Tämän jälkeen valitaan kysymys jolla aloitetaan indexin avulla. Otetaan tästä objektista tarvittu kysymys elementti ja siirretään se muuttujaan jota teksti elementti käyttää.

Myös vastaukset lisätään omaan listaansa, jotta ne ovat helppoja lisätä button elementteihin.

Ihan lopuksi shufflataan vielä myös kysymysten järjestys, jotta ne ovat eri järjestyksessä melkein joka kerta.

```
// Vertical Grid that is two columns wide
LazyVerticalGrid(
    cells = GridCells.Fixed( count: 2),
    horizontalArrangement = Arrangement.spacedBy(10.dp),
    verticalArrangement = Arrangement.spacedBy(10.dp),
    contentPadding = PaddingValues(horizontal = 10.dp, vertical = 4.dp),
    content = { this: LazyGridScope
        // Loops through answers and makes a button with the answer as text
        items(answers) { answer ->
            Button(
                modifier = Modifier.height(80.dp),
                // adds the first letter of the string inside of button, to user's score
                onClick =
                {
                    userScore.add(answer)
                    // check if there is a next question
                    if(promptCounter < QuizQuestions.size - 1) {
                        promptCounter += 1
                        return@Button
                    }
                    // if it is already last question, go to results screen
                    sendResults(userScore, correctAnswers)
                },
            )
            { this: RowScope
                Text(text = answer, textAlign = TextAlign.Start)
            }
        }
    })
}
}
```

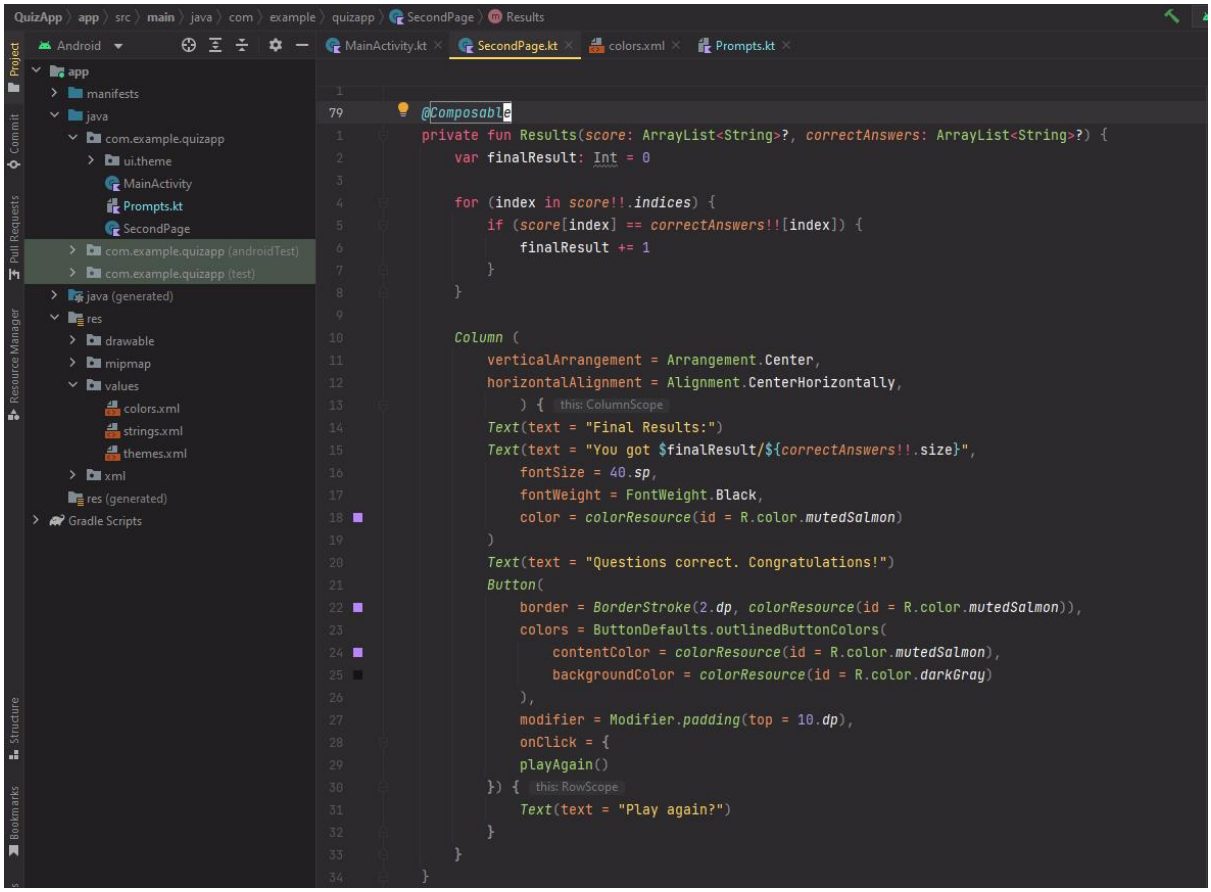
Käytin LazyVerticalGridiä, jotta sain kysymykset mukavasti grid muotoon, tässä lazy gridissä oli myös mahdollisuus käyttää argumenttia cells, joka mahdollisti, että rivissä oli aina kaksi elementtiä vierekkäin.

Tämä elementti looppaa sitten vastaukset listan, joille se rakentaa jokaiselle oman button elementin sekä onClick funktion.

Funktiossa käytin guard clause tapaa, jolla ohjelma tarkistaa, että onko kysymyksiä vielä jäljellä ja sitten vaihtaa kysymyksen ja palaa takaisin. Jos ei löydy toista kysymystä, laukaistaan sendResults funktio.

```
private fun sendResults(userScore: List<String>, correctAnswers: List<String>) {
    // initialize intent and put userScore into it
    val navigateForward = Intent( packageContext: this@MainActivity, SecondPage::class.java)
    navigateForward.putStringArrayListExtra( name: "userScore", ArrayList<String>(userScore))
    // Get correctAnswers and also put them in the Intent
    navigateForward.putStringArrayListExtra( name: "correctAnswers", ArrayList<String>(correctAnswers))
    // finally open results page
    startActivity(navigateForward)
}
```

sendResults() funktio tekee puolestaan intentin, johon sisällytetään käyttäjän pistemäärä sekä oikeat vastaukset.



Result näyttö on erillinen tiedosto ja erillinen compose kokonaan. Siellä näytetään vain tulokset numeroina ja luodaan nappula, jolla voidaan palata ensimmäiselle sivulle, jotta voi pelata uudestaan.

Tämä nappula ei palauta dataa Intentin mukana, jotta joka kierroksella peli alkaa puhtaalta lautaselta.

Mitä opin tehtävän aikana:

-Kuinka käyttää intenttiä Compose projektissa.

-Data classien rakentamisen Kotlinissa
Tämä oli erittäin helppoa, sillä classiin ei tarvinnut erillistä constructoria taikka muuttujia erikseen kirjoittaa classin sisälle.

-Gridin käyttöä composessa

-Että listojen ”shufflaaminen” on huomattavasti helpompaa kuin JavaScriptissä, jossa funktio piti rakentaa itse

-Kuinka käyttää remember muuttujia

-Paljon muuta pientä korjailua ja Kotlinin vahvuuksien hyväksi käyttämistä.