Сучасні технології мобільного програмування

Основні UI компоненти та управління станом

Слайди до лекцій (2 змістовий модуль)

Основні UI компоненти Jetpack Compose

- Компоненти Text, Button, Spacer, а також компонувальники Column та Row вже були розглянуті при реалізації навчального застосунку.
- Підготуємо екран для вивчення інших компонентів, який викликатимемо в MainActivity.

```
package ua.edu.znu.geoquizcomposeedu.ui.screens

import ...

@Composable
fun ComponentScreen(innerPadding: PaddingValues) {
//      Widget Code
   Box(
      modifier = Modifier
         .padding(innerPadding)
         .fillMaxSize(),
      contentAlignment = Alignment.Center
   ) {
//      Widget Code
      )
   }
}
```

```
@Preview(showSystemUi = true)
@Composable
fun ComponentScreenPreview() {
   ComponentScreen(
      innerPadding = PaddingValues(0.dp)
   )
}
```

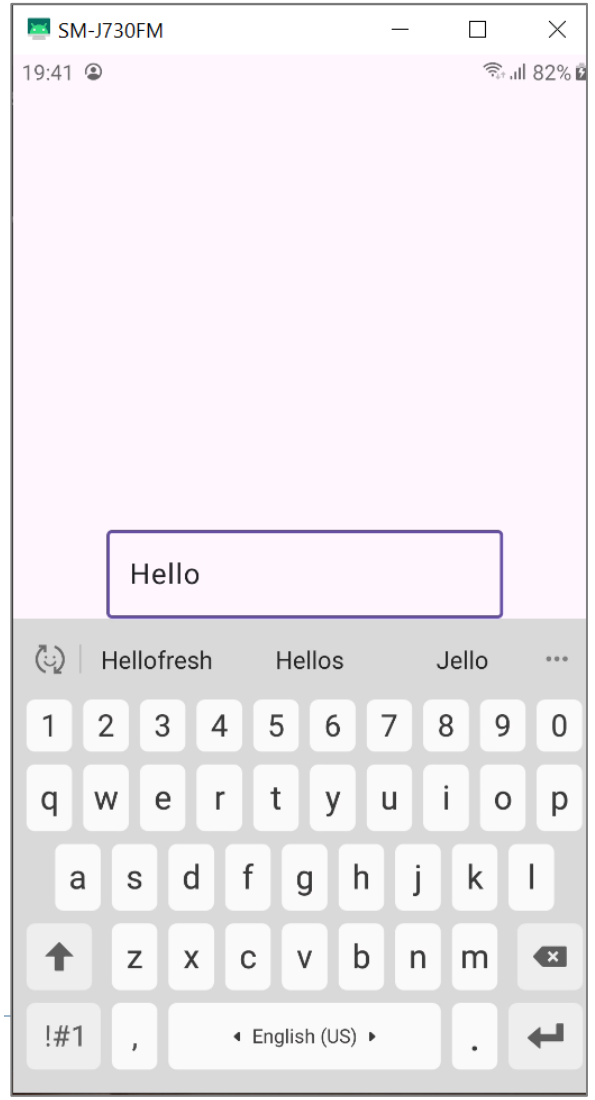
Компоненти Jetpack Compose

- Розглянемо компоненти текстового введення - TextField та OutlinedTextField.

```
@Composable
fun ComponentScreen(innerPadding: PaddingValues) {
    var textValue: String by remember {
        mutableStateOf("")
    }
    Box( ...) {
//      TextField(
        OutlinedTextField(
//          value = "qwerty",
            value = textValue,
            onValueChange = { newTextValue ->
                /* Handle text change */
                textValue = newTextValue
            }
        )
    }
}
```

Widget's State

Can't change without state



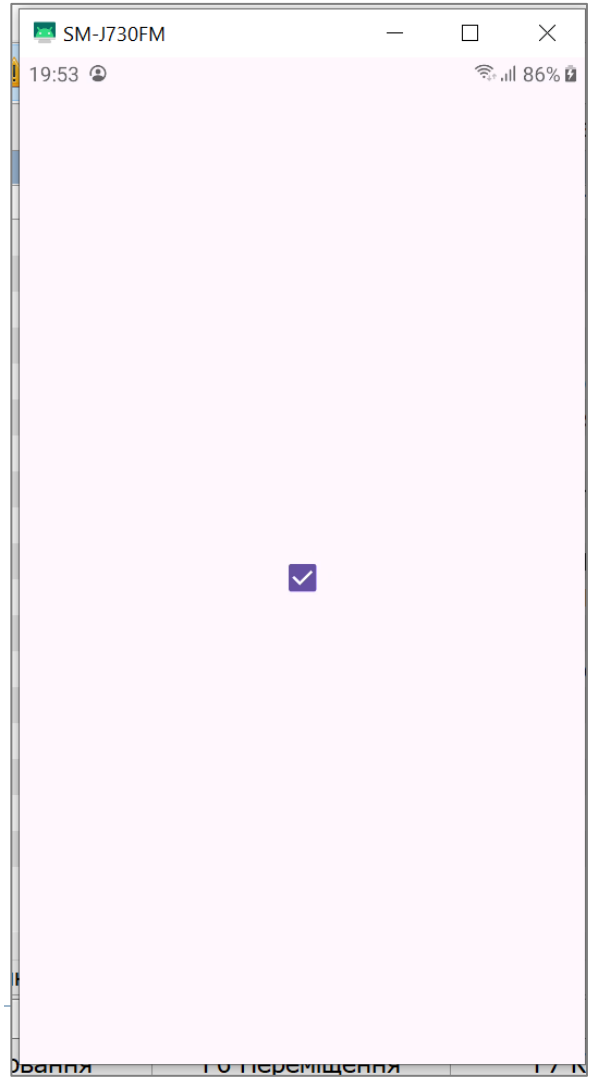
Компоненти Jetpack Compose

- Розглянемо компонент Checkbox.

```
@Composable
fun ComponentScreen(innerPadding: PaddingValues) {
    var state: Boolean by remember {
        mutableStateOf(false)
    }
    Box(...) {
        Checkbox(
//          checked = true,
            checked = state,
            onCheckedChange = { newCheckedState ->
                /* Handle checkbox state change */
                state = newCheckedState
            }
        )
    }
}
```

Widget's State

Can't change without state



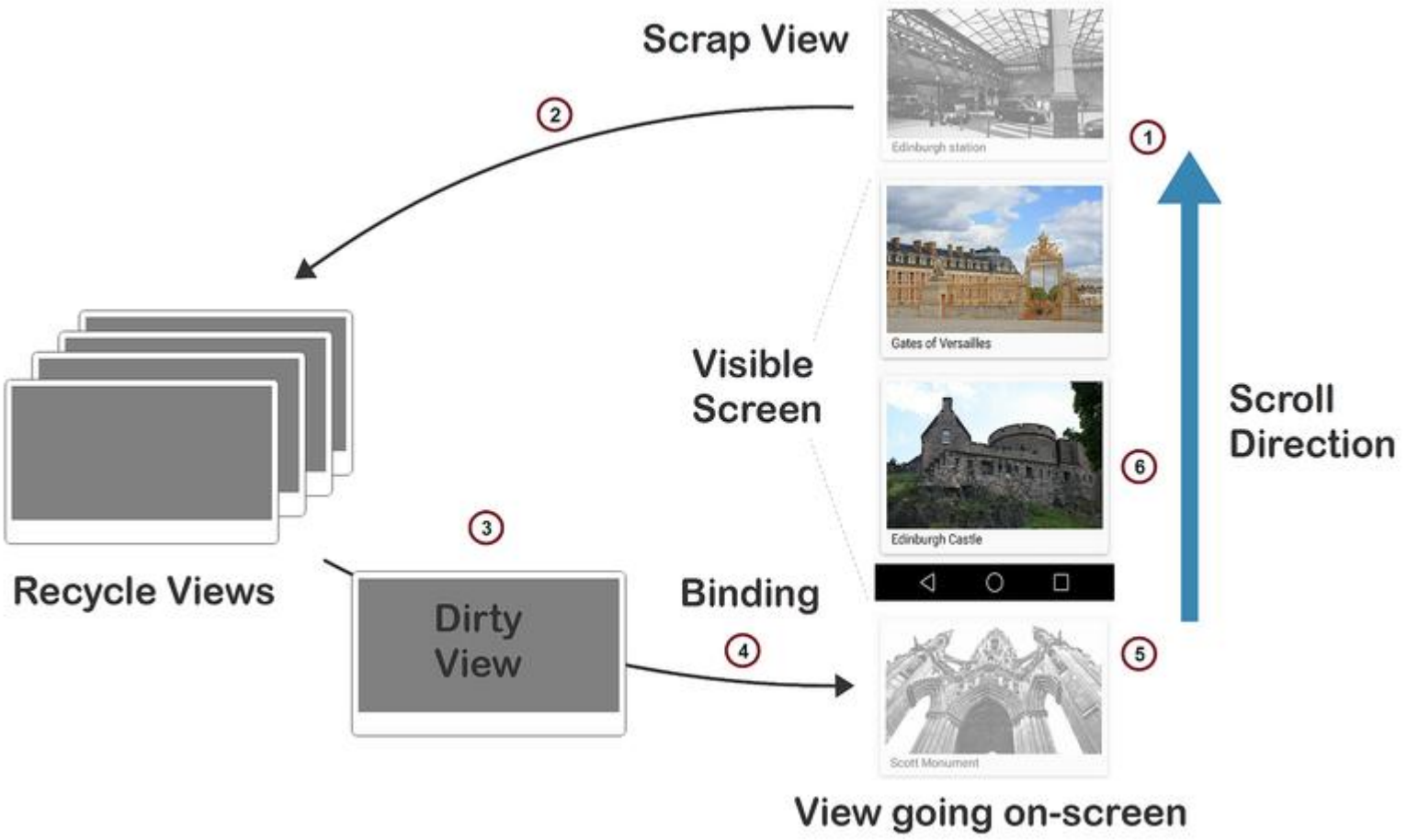
Компонент LazyColumn - організація списків

- Списки з прокруткою - найзручніші компоненти для відображення великої кількості інформації (малі екрани роблять некомфортною роботу з таблицями).
- Застосунки з екранами типу "список-деталізація" є одними з найрозповсюдженіших застосунків.
- Компоненти Android Views & XML для організації списків: ListView, Spinner, RecylerView (99%).



Компонент LazyColumn - організація списків

- Принцип роботи RecyclerView - повторне використання Views через ViewHolders, які виводять дані на екран



Компонент LazyColumn - організація списків

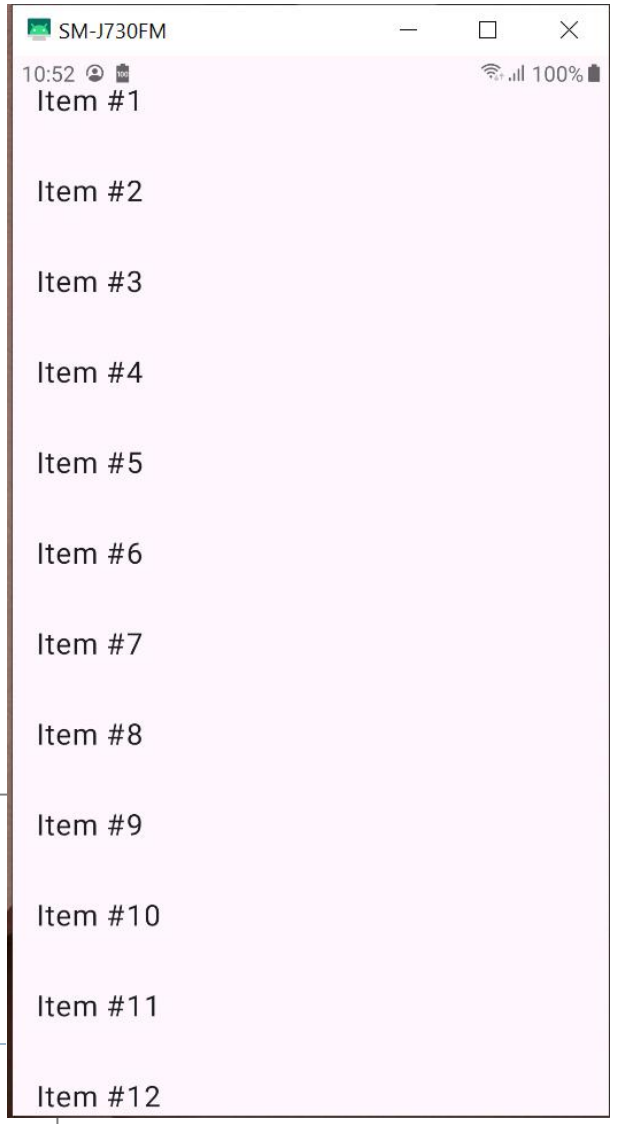
- Простий список

```
package ua.edu.znu.geoquizcomposeedu.educational

import ...

@Composable
fun SimpleListExample(innerPadding: PaddingValues) {
    LazyColumn(
        modifier = Modifier.fillMaxWidth()
    ) {
        items(100) { index ->
            Text(
                text = "Item #${index + 1}",
                modifier = Modifier.padding(16.dp)
            )
        }
    }
}
```

```
@Preview(showBackground = true)
@Composable
fun SimpleListExamplePreview() {
    SimpleListExample(innerPadding =
PaddingValues(0.dp))
}
```



Компонент LazyColumn - організація списків

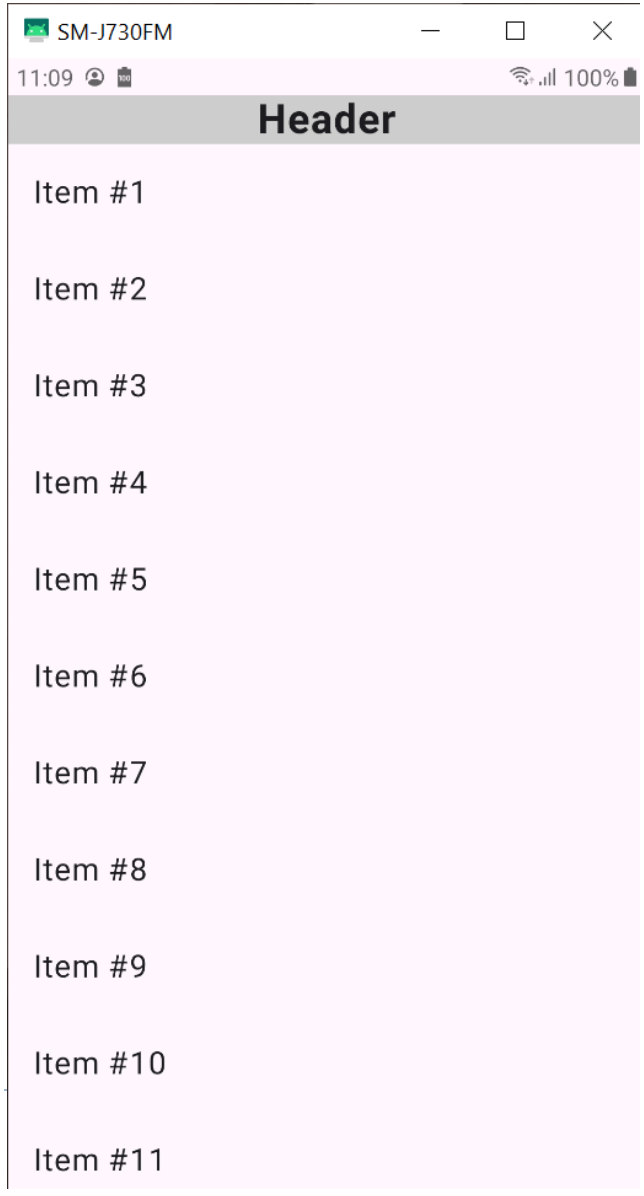
- Простий список з заголовком

```
package ua.edu.znu.geoquizcomposeedu.educational

import ...

@Composable
fun SimpleListExample(innerPadding: PaddingValues) {
    LazyColumn(
        modifier = Modifier.fillMaxWidth()
    ) {
        item {
            Text(
                text = "Header",
                fontWeight = FontWeight.Bold,
                fontSize = 24.sp,
                textAlign = TextAlign.Center,
                modifier = Modifier
                    .padding(top = 24.dp)
                    .fillMaxWidth()
                    .background(Color.LightGray)
            )
        } ...
```

Існує можливість множинного використання функцій item та items



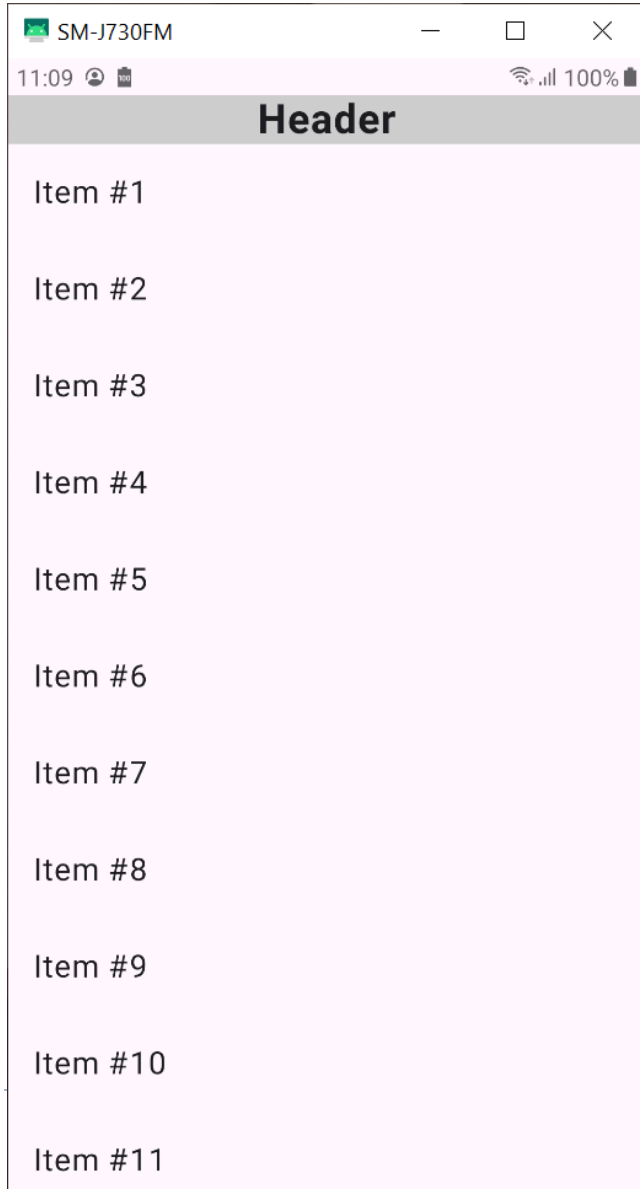
Компонент LazyColumn - організація списків

- Список з елементами List<String>

```
package ua.edu.znu.geoquizcomposeedu.educational

import ...

@Composable
fun SimpleListExample(innerPadding: PaddingValues) {
    val list: List<String> = remember {
        List(100) { index -> "Item #${index + 1}" }
    }
    LazyColumn(
        modifier = Modifier.fillMaxWidth()
    ) {
        item {
    .. }
        items(list) { item ->
            Text(
                text = item,
                modifier = Modifier.padding(16.dp)
            ) } } }
```



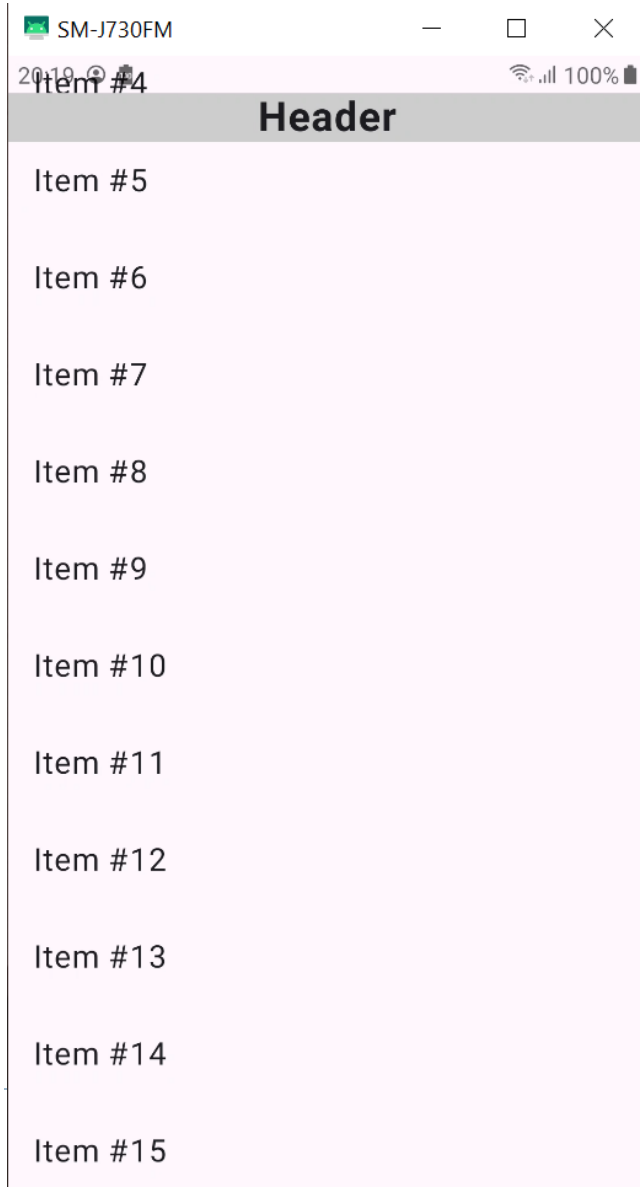
Компонент LazyColumn - організація списків

- Список з заголовком stickyHeader

```
package ua.edu.znu.geoquizcomposeedu.educational

import ...

@OptIn(ExperimentalFoundationApi::class)
@Composable
fun SimpleListExample(innerPadding: PaddingValues) {
    LazyColumn(
        modifier = Modifier.fillMaxWidth()
    ) {
        stickyHeader {
            Text(
                text = "Header",
                fontWeight = FontWeight.Bold,
                fontSize = 24.sp,
                textAlign = TextAlign.Center,
                modifier = Modifier
                    .padding(top = 24.dp)
                    .fillMaxWidth()
                    .background(Color.LightGray)
```



Компонент LazyColumn - організація списків

- Декілька списків з декількома stickyHeader

```
package ua.edu.znu.geoquizcomposeedu.educational

import ...

@OptIn(ExperimentalFoundationApi::class)
@Composable
fun SimpleListExample(innerPadding: PaddingValues) {
    val list: List<String> = remember {
        List(100) { index -> "Item #${index + 1}" }
    }
    LazyColumn(
        modifier = Modifier.fillMaxWidth()
    ) {
        stickyHeader {
            ListHeader("Header 1")
        }

        items(list) { item ->
            ListItem(text = item)
        } ...
```

1 список

Компонент LazyColumn - організація списків

```
...
        stickyHeader {
            ListHeader("Header 2")
        }

        items(50) { index ->
            ListItem(text = "Another Item #${index + 1}")
        }
    }
}

@Composable
fun ListHeader(text: String) {
    Text(
        text = text,
        fontWeight = FontWeight.Bold,
        fontSize = 24.sp,
        modifier = Modifier.padding(16.dp).fillMaxWidth()
            .background(Color.LightGray)     )
} ...
```

2 список

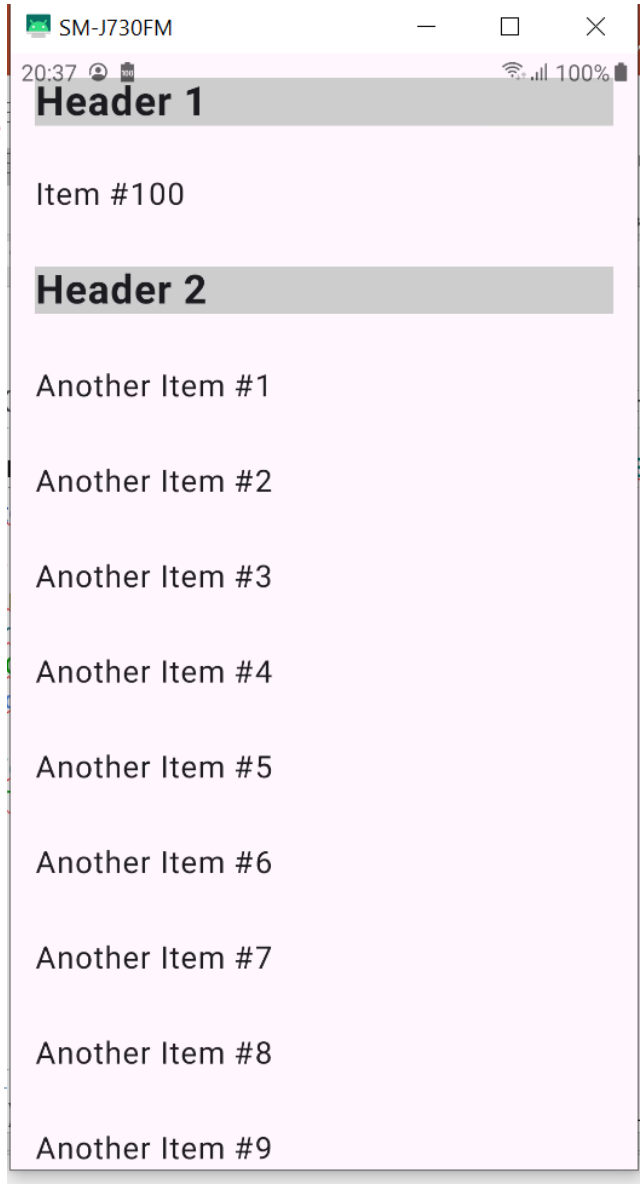
користувацька Composable

Компонент LazyColumn - організація списків

```
...
@Composable
fun ListItem(text: String) {
    Text(
        text = text,
        modifier = Modifier.padding(16.dp)
    )
}

@Preview(showBackground = true)
@Composable
fun SimpleListExamplePreview() {
    val innerPadding = PaddingValues(16.dp)
    SimpleListExample(innerPadding = innerPadding)
}
```

користувацька Composable



Компонент LazyColumn - організація списків

- Використання функцій-розширень для спрощення коду

```
package ua.edu.znu.geoquizcomposeedu.educational

import ...

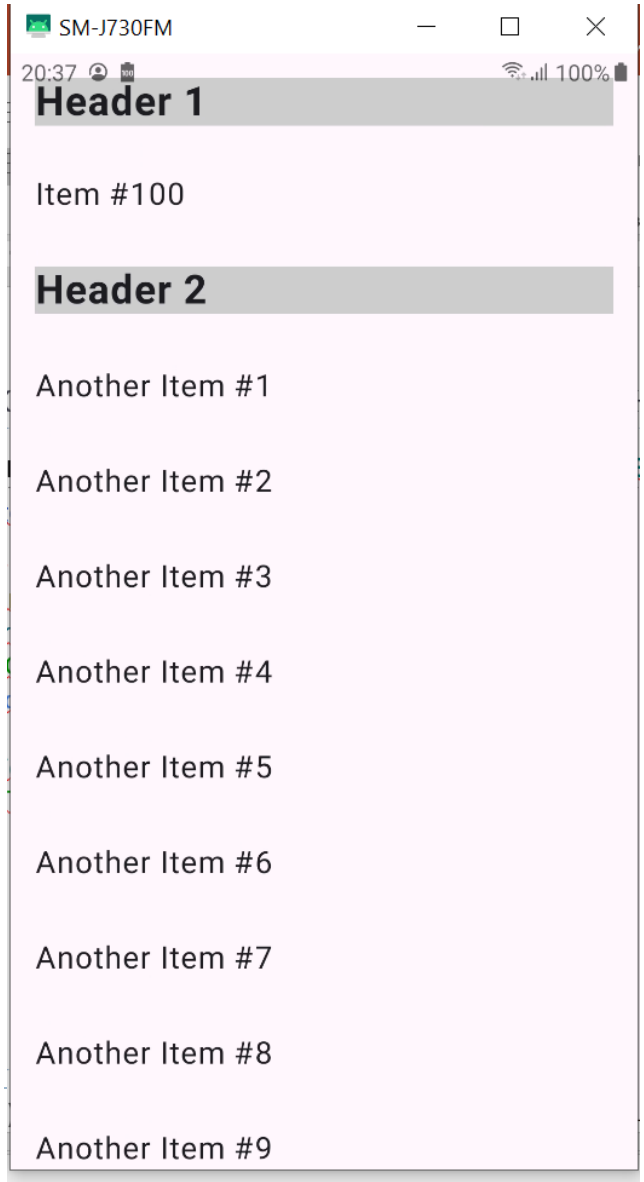
@OptIn(ExperimentalFoundationApi::class)
@Composable
fun SimpleListExample(innerPadding: PaddingValues) {
    val list1: List<String> = remember {
        List(100) { index -> "Item #${index + 1}" }
    }
    val list2: List<String> = remember {
        List(50) { index -> "Another Item #${index + 1}" }
    }
    LazyColumn(
        modifier = Modifier.fillMaxWidth()
    ) {
        listHeader("Header 1")
        listItems(list1)
        listHeader("Header 2")
        listItems(list2)     } } ...
```

Компонент LazyColumn - організація списків

```
...
@OptIn(ExperimentalFoundationApi::class)
fun LazyListScope.listHeader(text: String) {
    stickyHeader {
        Text(
            text = text,
            fontWeight = FontWeight.Bold,
            fontSize = 24.sp,
            modifier = Modifier
                .padding(16.dp)
                .fillMaxWidth()
                .background(Color.LightGray)
        )
    }
}

fun LazyListScope.listItems(list: List<String>) {
    items(list) { item ->
        Text(
            text = item,
            modifier = Modifier.padding(16.dp)
        )   } }
```

Функції-розширення на інтерфейсі LazyListScope



Реалізація обробників подій

- Класичні підходи до побудови UI базувалися на ***імперативному програмуванні***, коли у відповідь на взаємодію користувача з віджетом викликалася функція обробки цієї взаємодії. Програмістом організовувалося «прослуховування» події взаємодії (наприклад, за допомогою шаблону проєктування Спостерігач (Observer)) та обробка цієї події з відображенням результату в якомусь віджеті UI.
- Jetpack Compose передбачає ***декларативне програмування*** UI, коли задачу відслідковування взаємодії користувача з віджетом та оновлення даних – результату цієї взаємодії в UI бере на себе фреймворк.
- Декларативне програмування UI у Jetpack Compose базується на понятті **стану (State)** (див. далі).



Реалізація обробників подій - кнопка Next

```
private const val TAG = "MainScreen"

@Composable
fun MainScreen(
    innerPadding: PaddingValues,
) {
    var currentIndex = 0

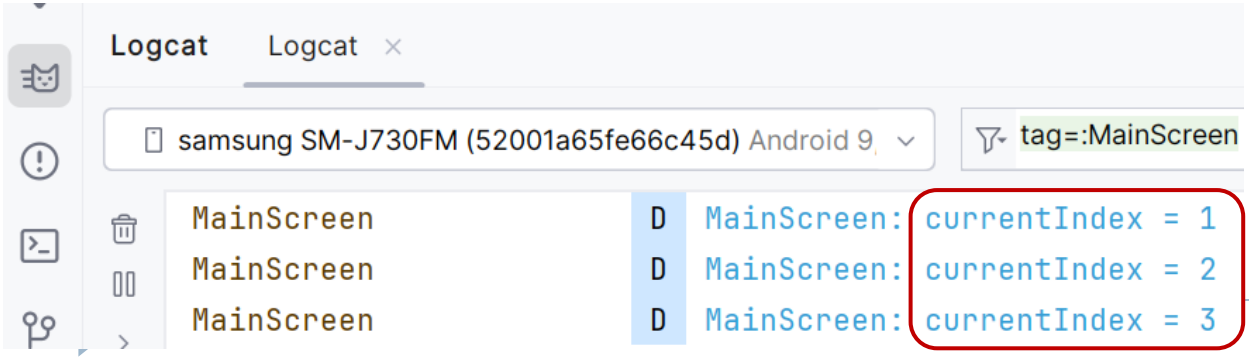
...
    Text(
        text = stringResource(id = questionBank[currentIndex].textResId),
        textAlign = TextAlign.Center,
    )
...
    Button(onClick = {
        currentIndex = ++currentIndex % questionBank.size
        Log.d(TAG, "MainScreen: currentIndex = $currentIndex")
    }) {
        Text(stringResource(id = R.string.next_button))
    }    }} ...
```



Реалізація обробників подій - кнопка Next

Перехід до нового питання після кліку на кнопку Далі не відбувається.
Для правильної роботи необхідно було б організовувати «прослуховування» події кліку по кнопці у компоненті Text.





Поняття стану

- За своєю суттю кожна програма працює з певними значеннями, які можуть змінюватись у часі. Такі значення можуть бути ***змінними***, ***об'єктами*** та їх ***властивостями***. Наприклад, взаємодія користувача з UI може призводити:
  - до інкременту значення лічильника, що відображається у текстовій мітці,
  - додавати нові дані до масиву або колекції, елементи якої відображаються у списку,
  - змінювати властивість об’єкта, дані якого відображаються у формі тощо.
- Такі змінювані величини, від яких залежить один або декілька віджетів UI (Composable функції) називається **станом** (State).
- Jetpack Compose надає засоби для організації відслідковуваного стану інтерфейсу і автоматично оновлює віджети (Composable функції), які використовують цей стан (залежать від цього стану).



Зміна стану

- Кожний UI компонент (віджет) є **незмінним** (**immutable**), тому після створення, наприклад, компонента **Text** з певним значенням властивості **text** змінити цю властивість не можна.
- Змінити властивості UI компонентf можна тільки використовуючи **стан.**
- Для створення змінюваного стану Jetpack Compose надає функцію ***mutableStateOf*****,** що приймає як аргумент ***початковий стан*** (значення змінної-стану або об'єкт, властивість якого визначає стан).
- У разі зміни стану Jetpack Compose виконує ***рекомпозицію*** (повторне виконання) Composable-функцій, які залежать від цього стану, автоматично оновлюючи інтерфейс новим значенням.
- Як правило змінна або об'єкт стану оголошуються у (або передаються до) Composable-функції, що відповідає за UI. При цьому виклик функції ***mutableStateOf*** зазначається як аргумент функції remember, яка забезпечує збереження значення стану при рекомпозиції Composable-функції.



Робота зі станом

```
private const val TAG = "MainScreen"

@Composable
fun MainScreen(
    innerPadding: PaddingValues,
) {
//    var currentIndex = 0
        val currentIndexState = remember { mutableStateOf(currentIndex) }

        val onNextButtonClick = () ->
                { currentIndexState.value = (currentIndexState.value + 1) % questionBank.size
                  Log.d(TAG, "MainScreen: currentIndex = ${currentIndexState.value}")
                 }
...
        Text(
            text = stringResource(id = questionBank[currentIndexState.value].textResId),
            textAlign = TextAlign.Center,
        )
...
        Button(onClick = onNextButtonClick) {
            Text(stringResource(id = R.string.next_button))        }    }} ...
```

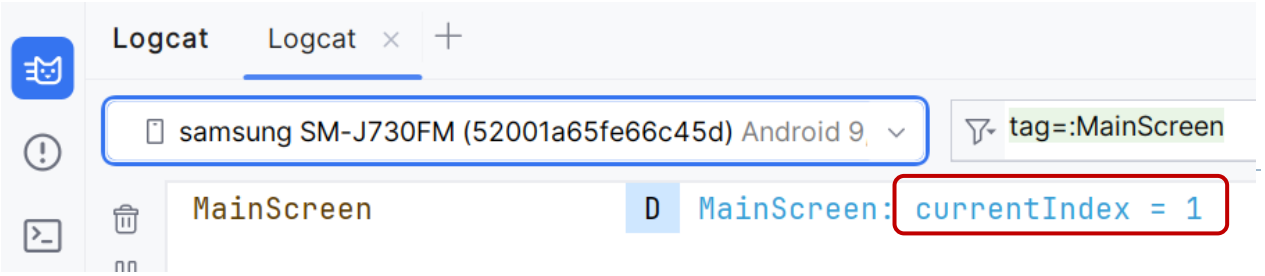
Оголошення стану Composable-функції MainScreen

Віджет Text залежить від стану

Робота зі станом

Відбувся перехід до нового питання після кліку на кнопку Далі





Робота зі станом

- Jetpack Compose, окрім *mutableStateOf*, надає для створення стану функції:
  - *mutableIntStateOf*(value: Int): MutableIntState
  - *mutableLongStateOf*(value: Long): MutableLongState
  - *mutableFloatStateOf*(value: Float): MutableFloatState
  - *mutableDoubleStateOf*(value: Double): MutableDoubleState
- Для доступу до параметрів, що відслідковуються станами, використовується поле типValue, наприклад, intValue

```
val currentIndexState = remember {mutableIntStateOf(currentIndex) }
val onNextButtonClick = () ->
        { currentIndexState.intValue = (currentIndexState.intValue + 1) % questionBank.size
          Log.d(TAG, "MainScreen: currentIndex = ${currentIndexState.intValue}")
         }
```

...

Підвищення ефективності уникненням автоупакування-авторозпакування

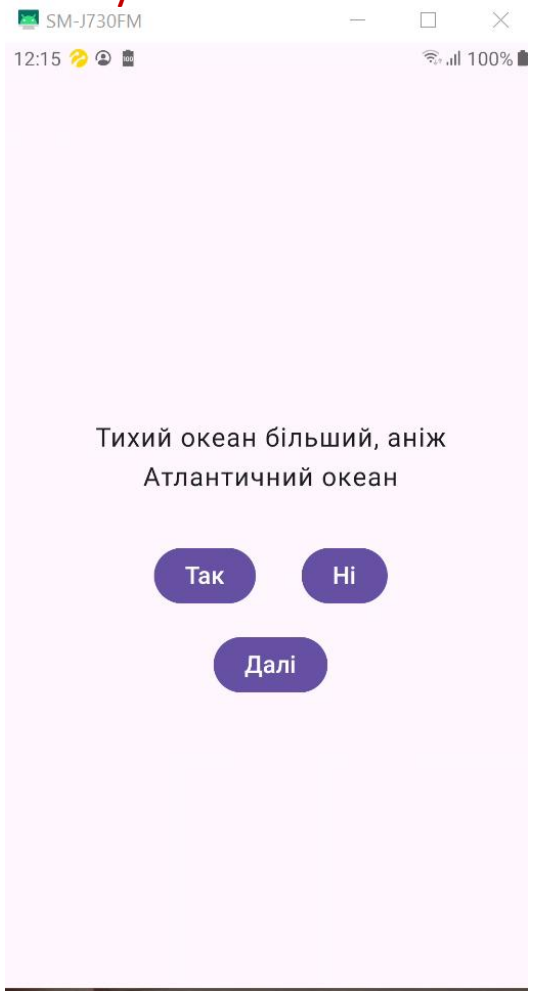


Робота зі станом

```
...
Text(
    text = stringResource(id = questionBank[currentIndexState.intValue].textResId),
    textAlign = TextAlign.Center,
)

Button(onClick = onNextButtonClick {
    Text(stringResource(id = R.string.next_button))
    }   }} ...
```

Вирівнювання тексту



Робота зі станом – клас стану мутабельний

- Часто стан може зберігати більш, аніж одну змінну, у такому разі доцільно розробити клас даних для стану:

```
data class MainScreenState(
    var currentIndex: Int = 0
)
@Composable
fun MainScreen(
    innerPadding: PaddingValues,
) {
    ...
val mainScreenState = remember { mutableStateOf(MainScreenState()) }
...
val onNextButtonClick: () -> Unit = {
        mainScreenState.value.currentIndex = (mainScreenState.value.currentIndex + 1)
                                              % questionBank.size
}
...
```

Зараз у класі стану для спрощення тільки одна МУТАБЕЛЬНА змінна

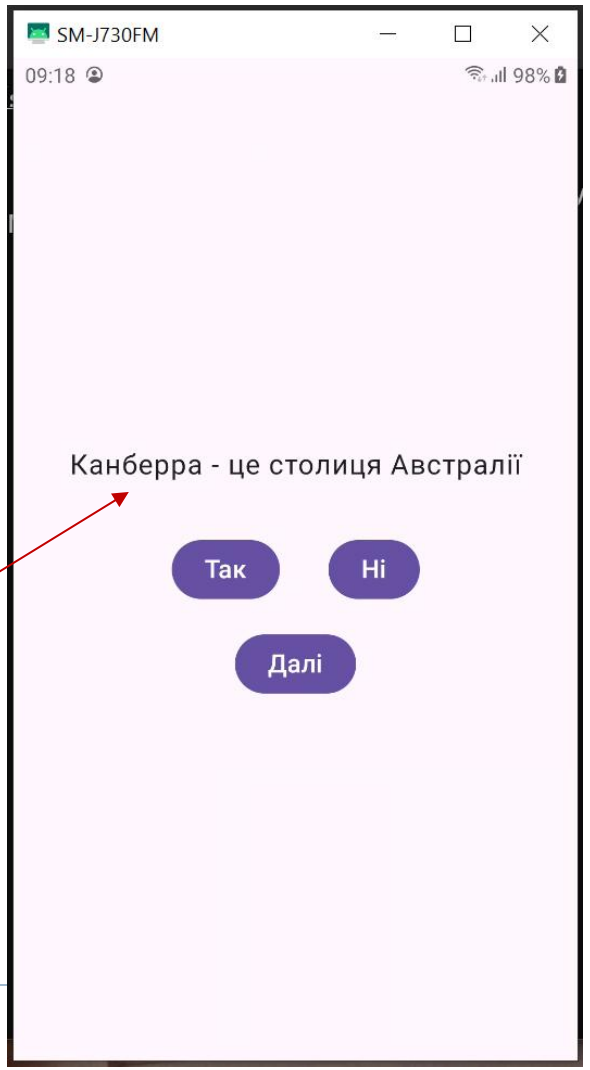
Тип змінної MutableState<MainScreenState>



Робота зі станом – клас стану мутабельний

```
...
Text(
    text = stringResource(id =
            questionBank[mainScreenState.value.currentIndex]
                                .textResId),
        )
```

Перехід до нового питання після кліку на кнопку Далі не відбувається.
Jetpack Compose «не бачить» зміни об’єкту стану, оскільки посилання на об’єкт стану не змінюється.



Робота зі станом – клас стану немутабельний

- Зробимо клас стану **немутабельним**, оголосивши як **val** його поля:

```
data class MainScreenState(
    val currentIndex: Int = 0
)
@Composable
fun MainScreen(
    innerPadding: PaddingValues,
) {
    ...
val mainScreenState = remember { mutableStateOf(MainScreenState()) }
...
val onNextButtonClick: () -> Unit = {
        val newMainScreenState = mainScreenState.value.copy(
            currentIndex = (mainScreenState.value.currentIndex + 1) % questionBank.size
        )
        mainScreenState.value = newMainScreenState
}
```

Зараз у класі стану для спрощення тільки одна НЕМУТАБЕЛЬНА змінна

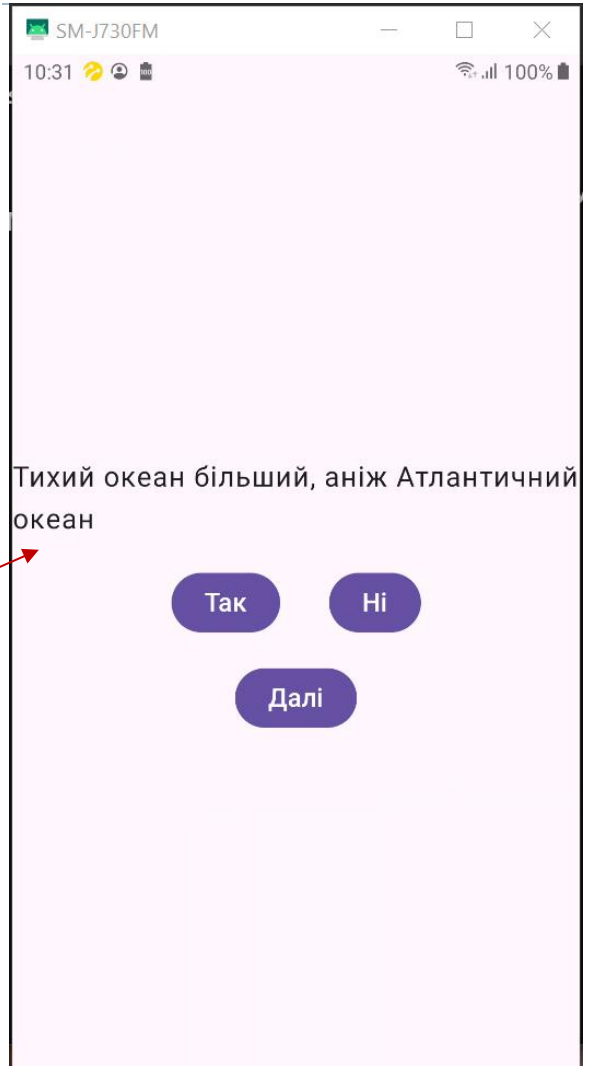
Тип змінної MutableState<MainScreenState>

Оскільки значення value немутабельне, новий стан створюються **копіюванням** старого з новим значенням currentIndex

Оновлення відслідковуваного стану НОВИМ ОБ'ЄКТОМ

Робота зі станом – клас стану немутабельний

```
...
Text(
   text = stringResource(id =
         questionBank[mainScreenState.value.currentIndex]
                        .textResId),
         textAlign = TextAlign.Center,
      )
```



Відбувся перехід до нового питання після кліку на кнопку Далі
Змінилося посилання на об’єкт стану, його «помітив» Jetpack Compose та виконав рекомпозицію.



Робота зі станом – покращення коду

- Мова Kotlin має механізми, що дозволяють спрощувати код:
  - деструктуризація (Destructuring Declarations);
  - делегування реалізації логіки геттера та сеттера властивості до іншого об'єкта (Delegated Properties)
- У разі використання деструктуризації оголошення стану може мати вигляд:

```
val (mainScreenState, setMainScreenState) = remember {
                                        mutableStateOf(MainScreenState()) }
```

- І у коді можна використовувати об'єкт mainScreenState та його сеттер:

```
Text(
   text = stringResource(id = questionBank[mainScreenState.currentIndex].textResId),
...
Button(onClick = {
        val newState = mainScreenState.copy(
            currentIndex = (mainScreenState.currentIndex + 1) % questionBank.size
        )
        setMainScreenState(newState)
        Log.d(TAG, "MainScreen: currentIndex = ${mainScreenState.currentIndex}")
...
```

Робота зі станом – покращення коду

- У разі використання делегування властивостей оголошення стану може мати вигляд (тепер це звичайна мутабельна змінна):

```
var mainScreenState by remember { mutableStateOf(MainScreenState()) }
```

Вираз by делегує об'єкту MutableState<MainScreenState> обов'язки читання та встановлення значення стану через змінну mainScreenState (тобто тепер вона містить mainScreenState.value)

- У коді можна використовувати її як звичайну змінну без явного сеттера:

```
Text(
   text = stringResource(id = questionBank[mainScreenState.currentIndex].textResId),
...
Button(onClick = {
        mainScreenState = mainScreenState.copy(
            currentIndex = (mainScreenState.currentIndex + 1) % questionBank.size
        )
        Log.d(TAG, "MainScreen: currentIndex = ${mainScreenState.currentIndex}")
...
```

Робота зі станом інтерфейса - поворот екрану

- Оскільки при повороті екрану активність запускається заново, буде знов показане перше питання.
- Для збереження індексу між поворотами слід замість функції **remember** використовувати функцію **rememberSaveable**:

```
var mainScreenState by rememberSaveable { mutableStateOf(MainScreenState()) }
```

- Ця функція автоматично зберігає значення змінної стану до об'єкту **android.os.Bundle**, який переживає видалення та повторне створення активності. Цей об'єкт передається як параметр до методу життєвого циклу активності

```
onCreate(savedInstanceState: Bundle?)
```

- Jetpack Compose автоматично відтворює значення змінної стану з об'єкта android.os.Bundle.
- Але у разі використання в якості аргументу **rememberSaveable** кастомного класу стану (MainScreenState) програм не запрацює, оскільки Jetpack Compose не знає як серіалізувати об'єкт MainScreenState для розміщення у Bundle

Робота зі станом інтерфейса - поворот екрану

- У разі, коли стан зберігається тільки ***в одній змінній стандартного типу (число або рядок)***, функція працювати буде, оскільки вона «знає» як серіалізувати такі типи:

```
// Use delegate getter and setter for state access
var currentIndexState by rememberSaveable { mutableIntStateOf(0) }

val onNextButtonClick: () -> Unit = {
    currentIndexState = (currentIndexState + 1) % questionBank.size
}

...
    Text(
        // Use delegate getter for state access
        text = stringResource(id = questionBank[currentIndexState].textResId),
        ...
    )
    ...
    Button(onClick = onNextButtonClick) {
        Text(stringResource(id = R.string.next_button))
    }  }}
```

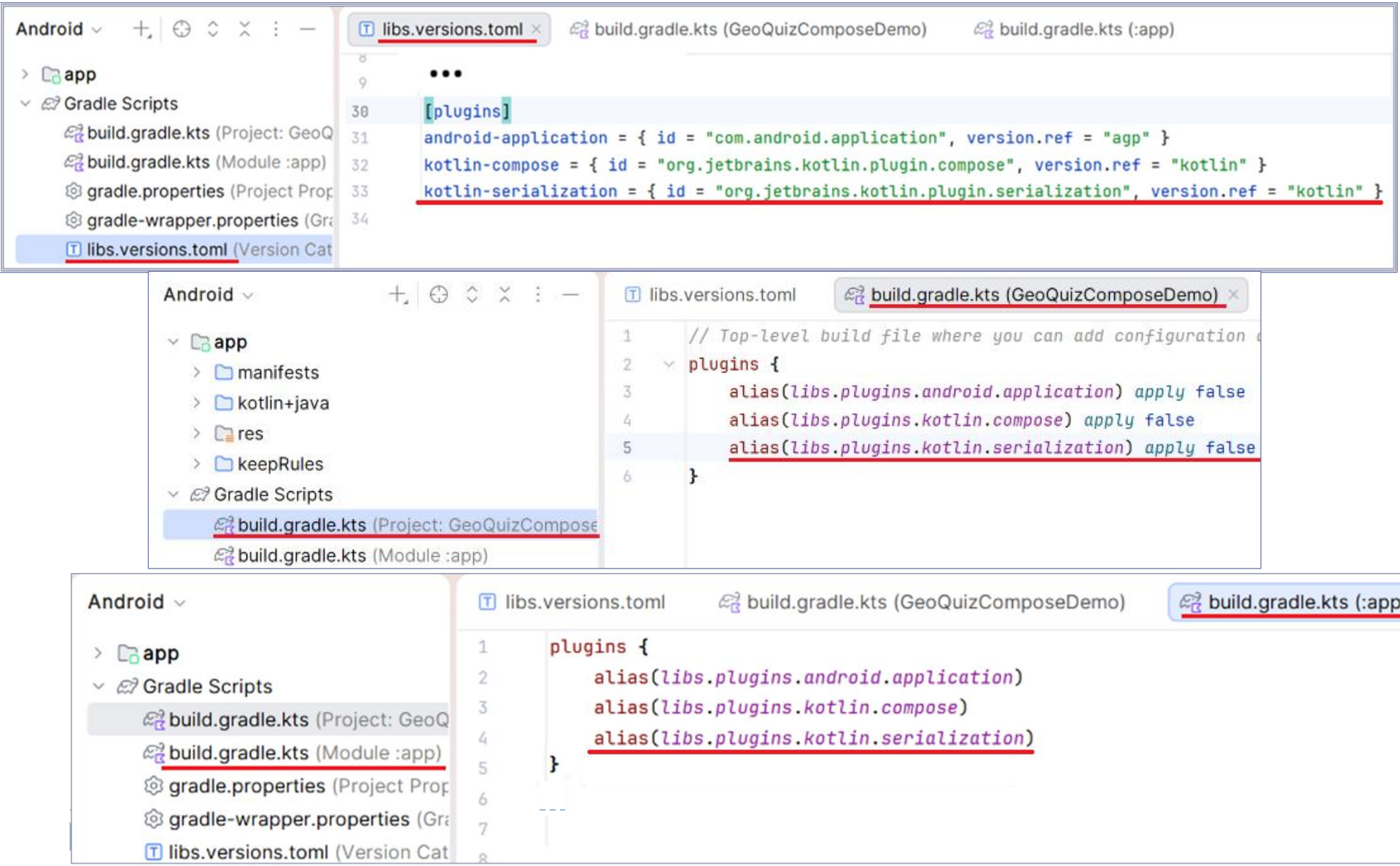
Робота зі станом інтерфейса - поворот екрану

- Якщо стан відслідковує декілька змінних і вони мають тип чисел або рядків, можна імплементувати користувацький дата клас стану від інтерфейсу Serializable для забезпечення зберігання об’єкту цього класу у Bundle.
- Для цього потрібно додати до проєкту плагін org.jetbrains.kotlin.plugin.serialization
  - Додати визначення плагіну та його версію до libs.versions.toml
  - Додати визначення плагіну до buld.gradle.kts проєкту
  - Додати визначення плагіну до buld.gradle.kts модуля.

див. далі



Додання плагіну та залежності для серіалізації



Робота зі станом інтерфейса - поворот екрану

```
data class MainScreenState(
    val currentIndex: Int = 0
) : Serializable
```

Імплементує інтерфейс **java.io.Serializable**

```
@Composable
fun MainScreen(
    innerPadding: PaddingValues,
) {
...
    var mainScreenState by rememberSaveable { mutableStateOf(MainScreenState()) }

     val onNextButtonClick: () -> Unit = {
            mainScreenState = mainScreenState.copy(
            currentIndex = (mainScreenState.currentIndex + 1) % questionBank.size
        )
    }

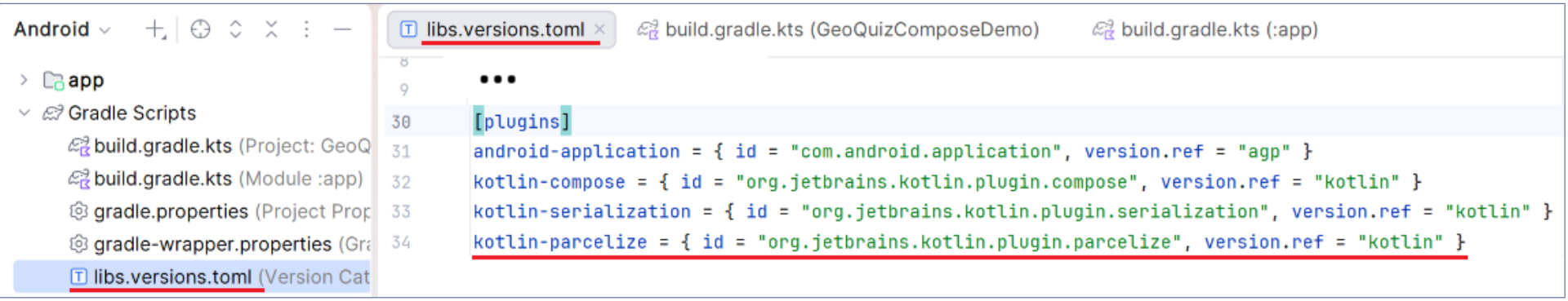
    ...
        Text(
            // Use delegate getter for state access
            text = stringResource(id = questionBank[mainScreenState.currentIndex].textResId),
...
```

Робота зі станом інтерфейса - поворот екрану

- На жаль збереження даних до Bundle за допомогою серіалізації є не дуже ефективним способом і при збереженні великих обсягів даних можуть виникнути проблеми.
- Вихід - використання **парселізації** - це оптимізований для Android механізм **серіалізації об'єктів**, який використовується для швидкої передачі складних даних між компонентами додатка.
- Порівняння Parcelable проти Serializable

Характеристика	Parcelable (Android)	Serializable (Java)
Швидкість	Дуже висока (працює в рази швидше)	Повільна (використовує рефлексію)
Споживання пам'яті	Мінімальне	Створює багато тимчасових об'єктів
Реалізація	Потребує плагіна або ручного коду	Достатньо лише імплементувати Serializable
Призначення	Тільки для передачі даних у пам'яті (IPC)	Для передачі в пам'яті, збереження у файли чи мережу

Додання плагіну для парселізації







Робота зі станом інтерфейса - поворот екрану

```
@Parcelize
data class MainScreenState(
    val currentIndex: Int = 0
) : Parcelable
```

Генерує реалізацію абстрактних методів інтерфейсу **android.os.Parcelable**

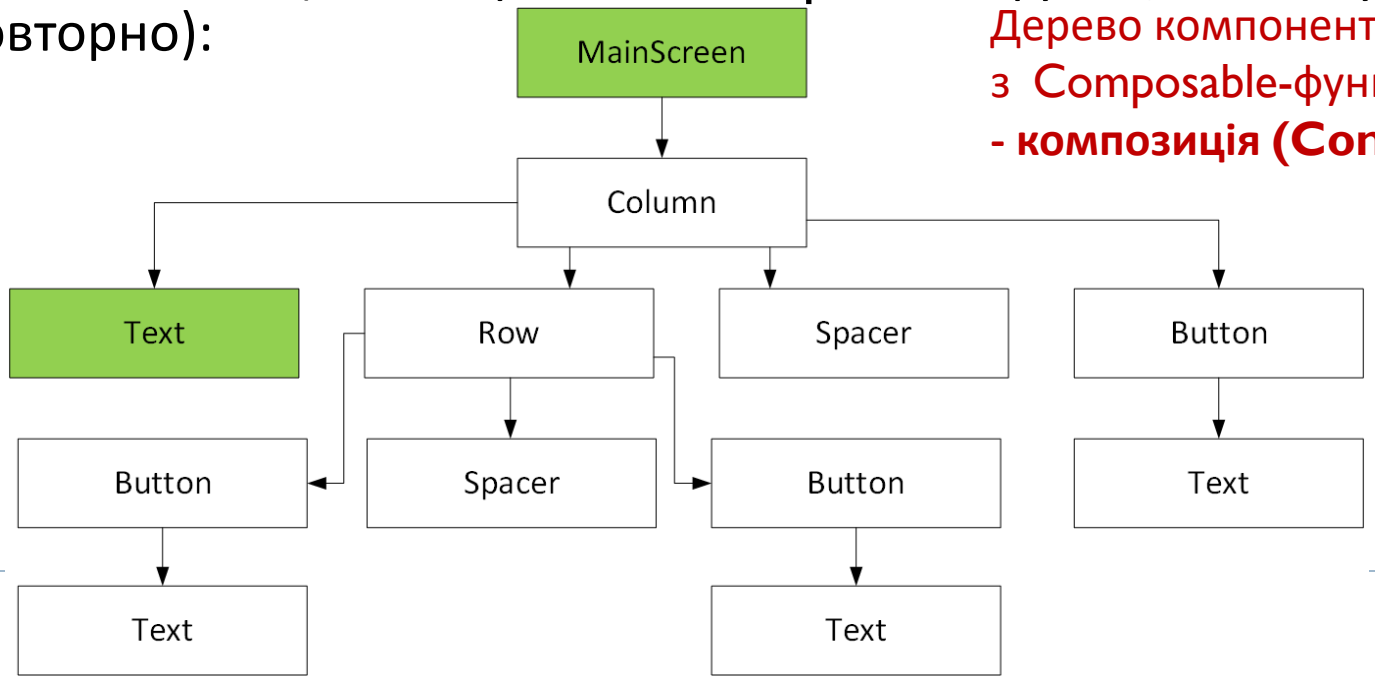
```
@Composable
fun MainScreen(
    innerPadding: PaddingValues,
) {
    ...
    var mainScreenState by rememberSaveable { mutableStateOf(MainScreenState()) }

     val onNextButtonClick: () -> Unit = {
            mainScreenState = mainScreenState.copy(
            currentIndex = (mainScreenState.currentIndex + 1) % questionBank.size
        )
    }

    ...
        Text(
            // Use delegate getter for state access
            text = stringResource(id = questionBank[mainScreenState.currentIndex].textResId),
...
```

Композиція у Jetpack Compose

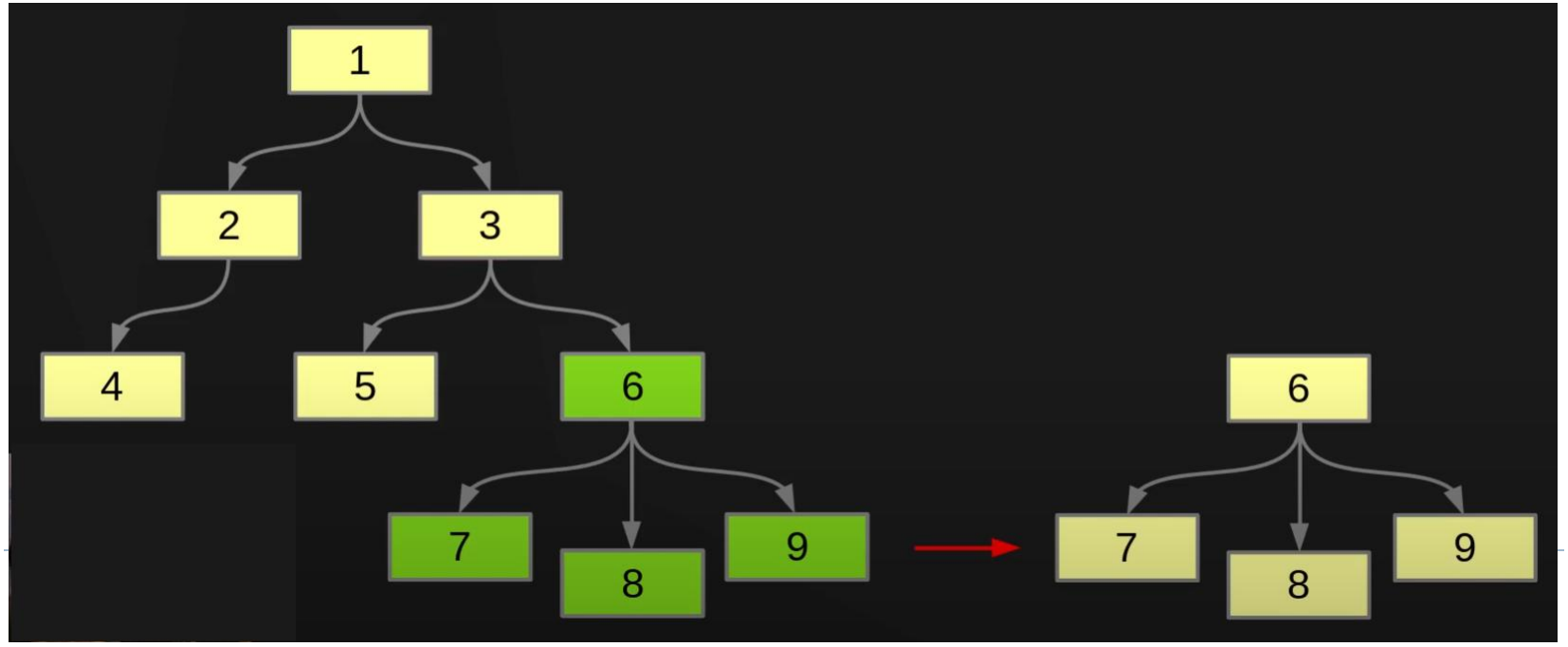
- **Композиція (Composition)** - це деревовидна структура інтерфейсу застосунку, побудована Composable-функціями.
- Оскільки компоненти інтерфейсу, створені Composable-функціями є незмінними, для їх зміни/оновлення необхідно використовувати стан.
- Зміна стану запускає процес **рекомпозиції**, який оновлює компоненти інтерфейсу (виконуючи Composable-функції ще раз).
- При цьому на інтерфейсі перемальовуються тільки ті частини дерева, які змінилися (хоча коренева Composable функція виконується повторно):



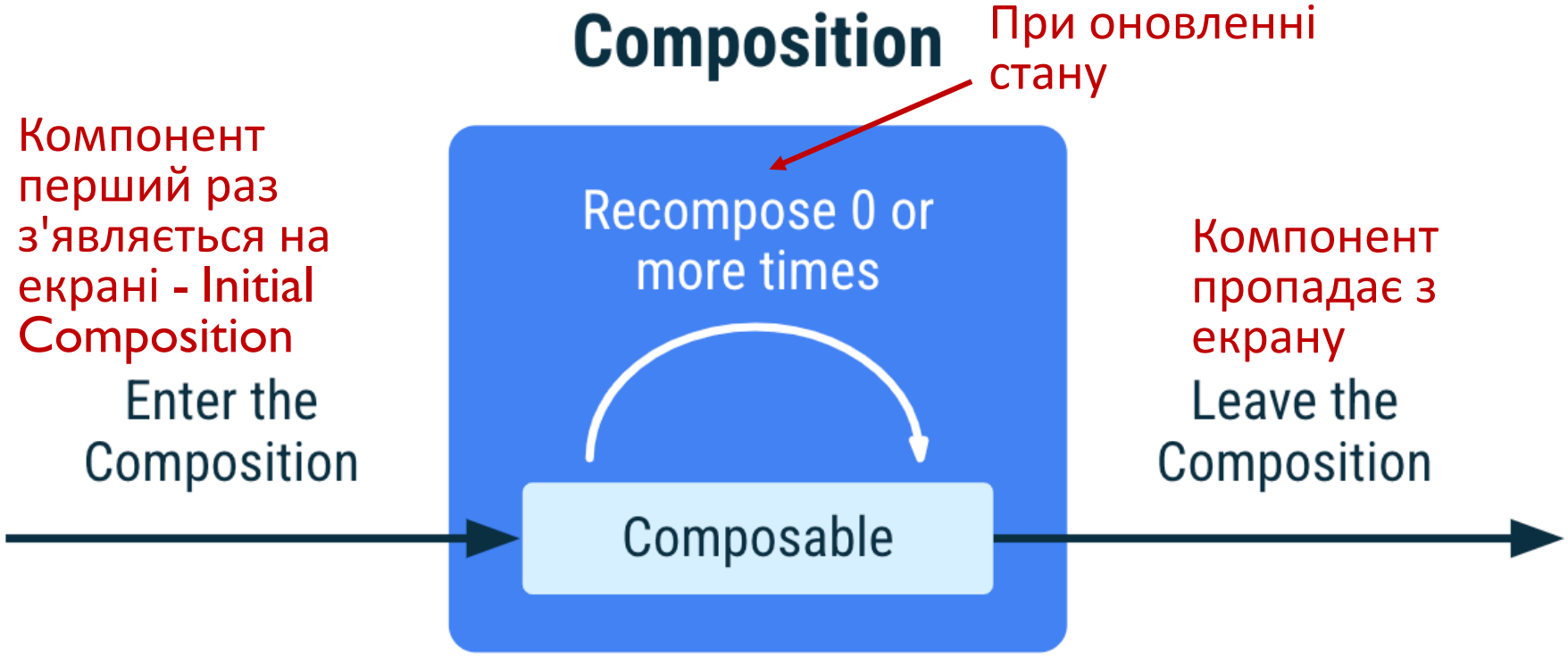
Дерево компонентів
з Composable-функцій
- композиція (Composition)

Композиція у Jetpack Compose

- Composable-функції не є композицією.
- Composable-функції обслуговують композицію (виконують дії, які впливають на композицію екрану).
- Необхідно розуміти, що компоненти Jetpack Compose - це не об'єкти, а результат виконання Composable функцій.
- Будь яка частина композиції теж є композицією (будь яка гілка дерева компонентів може розглядатися як дерево).



Життєвий цикл композиції



1. На першому етапі композиція створюється (коли внаслідок роботи Composable-функцій компонент вперше з'являється на екрані) - Initial Composition - називається Enter (вхід до композиції).
2. Опціонально композиція може бути оновлена 0 та більше разів (при оновленні стану) - називається Recomposition of Composition.
3. Композиція знищується, коли той або інший компонент пропадає з екрану - називається Leave - вихід з композиції.

Робота зі станом інтерфейса -динамічний компонент

- Життєвий цикл гілки дерева може бути коротшим цикла дерева

Файл LogLifecycle.kt

```
package ua.edu.znu.geoquizcomposeedu.util

import ...

private const val TAG = "CompositionLifecycle"

@Composable
fun logCompositionLifecycle(name: String): Any = remember {
    LifecycleRememberObserver(name)
}

private class LifecycleRememberObserver(
    private val name: String) : RememberObserver {

    override fun onRemembered() {
        Log.d(TAG, "$name.onEnter")
    }

    override fun onForgotten() {
        Log.d(TAG, "$name.onLeave")
    }

    override fun onAbandoned()=Unit
}
```

Клас, об’єкт якого буде сповіщений,
коли він буде задіяний у композиції та коли більше не буде у ній задіяний

Робота зі станом інтерфейса -динамічний компонент

```
@Composable
fun MainScreen(
    innerPadding: PaddingValues,
) {
    ...
    var mainScreenState by rememberSaveable { mutableStateOf(MainScreenState()) }

    logCompositionLifecycle("MainScreen")
    Column(
        ...
    ) {
...
        Button(onClick = onNextButtonClick
        ) {
            Text(stringResource(id = R.string.next_button))
        }
...
```

Логування композиції функції MainScreen



Робота зі станом інтерфейса -динамічний компонент

```
...
        Box(
            modifier = Modifier.height(100.dp)
        ) {
            if(mainScreenState.currentIndex == questionBank.size -1) {
                logCompositionLifecycle("LastQuestionText")
                Text(
                    text = "This is the last question"
                )
            }
        }
    }
}
```

Логування композиції функції Box

Logcat Logcat

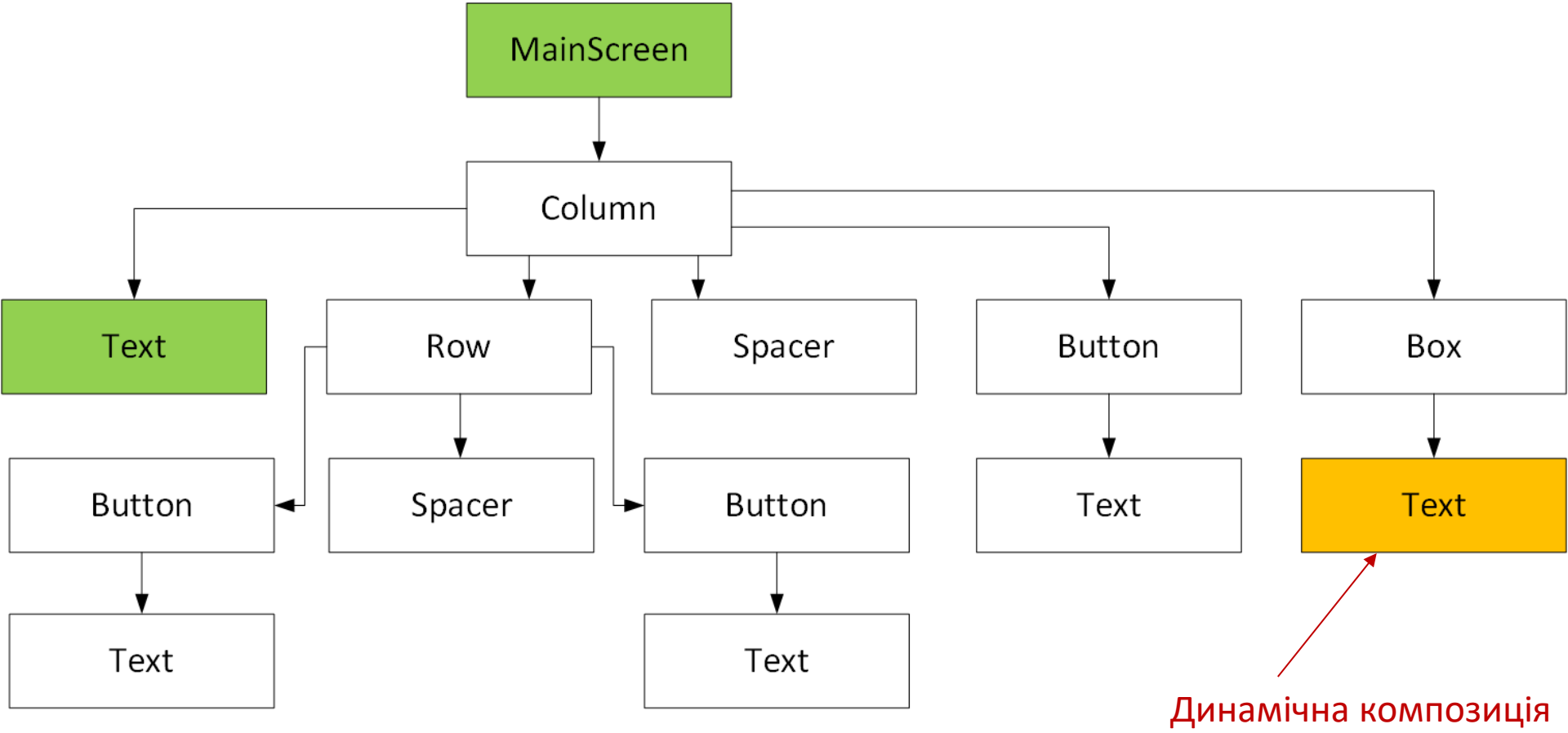
Medium Phone (emulator-5554) Android 17, API 3 tag:CompositionLifecycle

```
CompositionLifecycle  D  LastQuestionText.onEnter
CompositionLifecycle  D  LastQuestionText.onLeave
CompositionLifecycle  D  MainScreen.onLeave
CompositionLifecycle  D  MainScreen.onEnter
```

Останнє питання

Поворот екрану

Робота зі станом інтерфейса -динамічний компонент



Реалізація обробки подій відповідей на питання

```
package ua.edu.znu.geoquizcomposeedu.ui.screens

import ...

private const val TAG = "MainScreen"

@Parcelize
data class MainScreenState(
    val currentIndex: Int = 0
) : Parcelable

@Composable
fun MainScreen(
    innerPadding: PaddingValues,
) {
    val questionBank = listOf(
        Question(R.string.question_australia, true),
        Question(R.string.question_oceans, true),
        Question(R.string.question_mideast, false),
        Question(R.string.question_africa, false),
        Question(R.string.question_americas, true),
        Question(R.string.question_asia, true)
    )...
```

Реалізація обробки подій відповідей на питання

```
var mainScreenState by rememberSaveable { mutableStateOf(MainScreenState()) }
val context = LocalContext.current
val onNextButtonClick: () -> Unit = {
    mainScreenState = mainScreenState.copy(
        currentIndex = (mainScreenState.currentIndex + 1) % questionBank.size
    )
    Log.d(TAG, "onNextButtonClick: currentIndex = ${mainScreenState.currentIndex}")
}
val onAnswerButtonClick: (isTrue: Boolean) -> Unit = { isTrue ->
    val correctAnswer = questionBank[mainScreenState.currentIndex].answer
    val messageResId = if (isTrue == correctAnswer) {
        R.string.correct_toast
    } else {
        R.string.incorrect_toast
    }
    Toast.makeText(context, messageResId, Toast.LENGTH_SHORT).show()
}
```

Контекст функції MainScreen для компонента Toast

Універсальний обробник кліку по кнопкам відповідей з перевіркою правильності відповіді

Також залежить від стану

Реалізація обробки подій відповідей на питання

```
...
...
//   logCompositionLifecycle("MainScreen")
   Column(
      modifier = Modifier
         .fillMaxSize()
         .padding(innerPadding),
      verticalArrangement = Arrangement.Center,
      horizontalAlignment = Alignment.CenterHorizontally
   ) {
      Text(
         text = stringResource(id = questionBank[mainScreenState.currentIndex].textResId),
         textAlign = TextAlign.Center,
      )
...
```

Контекст функції MainScreen для компонента Toast



Реалізація обробки подій відповідей на питання

```
...
    Row(
        modifier = Modifier.size(width = 200.dp, height = 90.dp),
        horizontalArrangement = Arrangement.SpaceEvenly,
        verticalAlignment = Alignment.CenterVertically
    ) {
        Button(
            onClick = { onAnswerButtonClick(true) }
        ) {
            Text(stringResource(id = R.string.true_button))
        }
        Spacer(modifier = Modifier.width(16.dp))
        Button(
            onClick = { onAnswerButtonClick(false) }
        ) {
            Text(stringResource(id = R.string.false_button))
        }
        Spacer(modifier = Modifier.width(16.dp))
...
```



Реалізація обробки подій відповідей на питання

```
...
        Button(
            onClick = onNextButtonClick
        ) {
            Text(stringResource(id = R.string.next_button))
        }
    }
}
```

