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)
   )
}
```

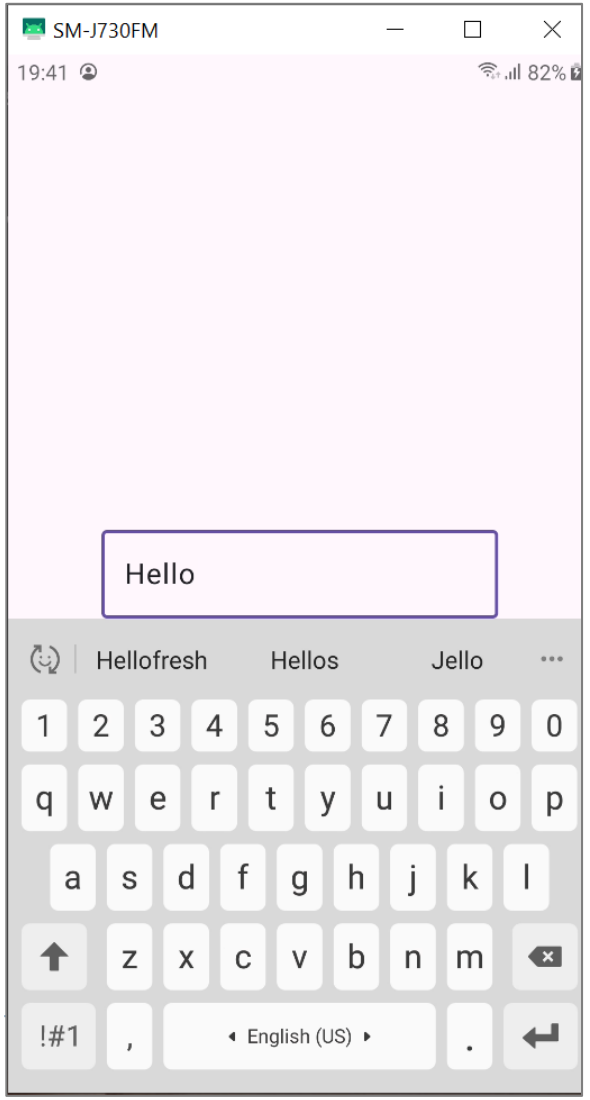
Компоненти Jetpack Compose

- 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



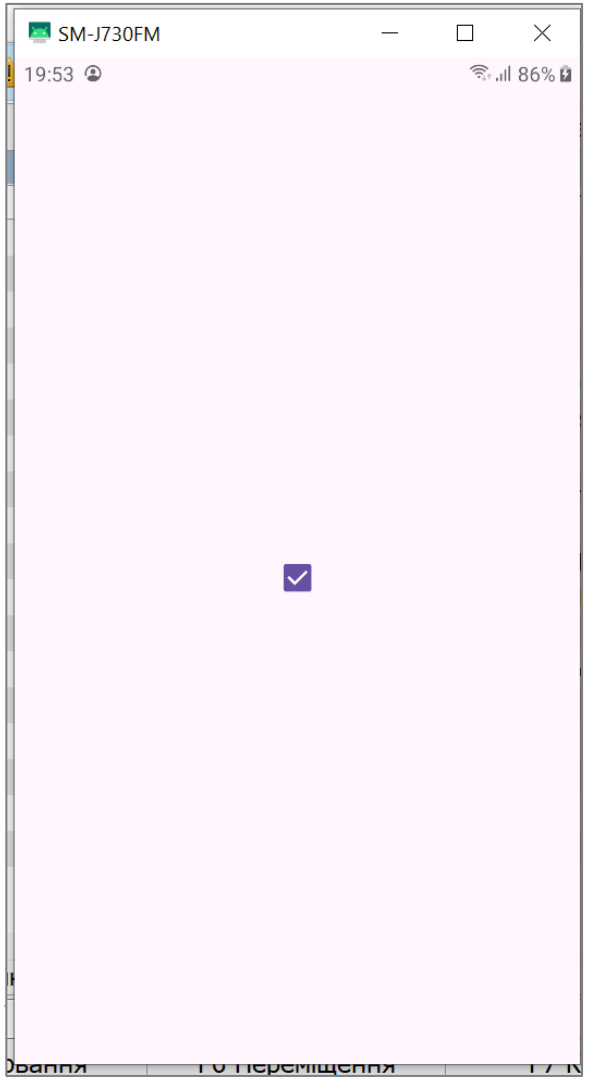
Компоненти Jetpack Compose

- 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



Реалізація обробників подій - кнопка Next

```
private const val TAG = "MainScreen"
@Composable
fun MainScreen(
    innerPadding: PaddingValues,
) {
...
        Text(
            text = stringResource(id = questionBank[currentIndex].textResId),
            modifier = Modifier.padding(16.dp)
        )
...
        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

Поняття стану

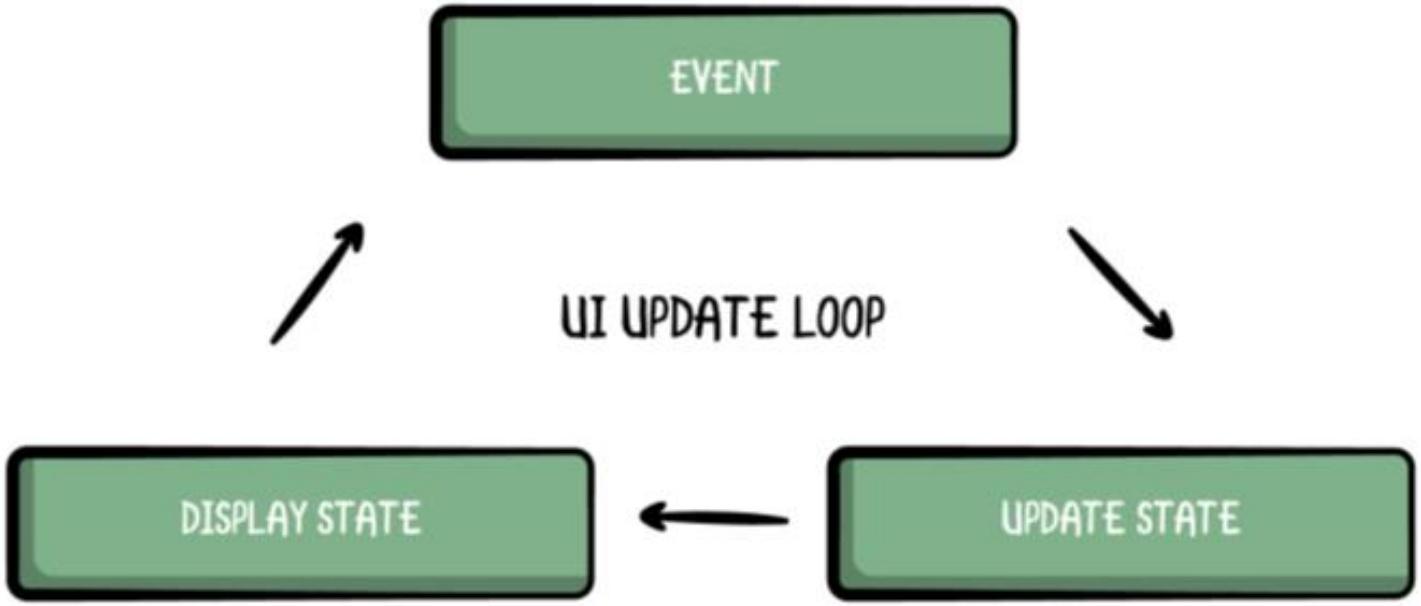
- За своєю суттю кожна програма працює з певними значеннями, які можуть змінюватись у часі. Такі значення можуть бути як об'єктами, так і властивостями об'єктів. Наприклад, можуть додаватися нові об'єкти, з якими працює застосунок, заповнюючи дані об'єктів, можуть зчитуватися та змінюватися дані таких об'єктів, видалятися певні дані об'єктів або самі об'єкти тощо.
- **Стан** – будь-яка властивість об'єкту, з яким працює застосунок, що може змінюватися ід час роботи застосунку. Ці значення можуть включати будь-що: від запису у базі даних до властивості класу.
- При зміні значення хоча б однієї з властивостей об'єкту говорять про **зміну стану**.
- При зміні стану необхідно оновлювати інтерфейс користувача, щоб відобразити ці зміни (Jetpack Compose це робить автоматично).

Робота зі станом інтерфейса

- Кожний UI компонент (віджет) є незмінним (immutable), тому після створення, наприклад, компонента **Text** з певним значенням властивості **text** змінити цю властивість не можна.
- Змінити компонент можна використовуючи **стан (state)** - це дані, що визначають як виглядає інтерфейс у конкретний момент.
- При зміні таких даних, компоненти, які від них залежать, будуть оновлені автоматично (буде повторно виконаний код їх функцій, але вже з новими значеннями даних, що визначають стан).
- Для створення стану необхідно додати **val** змінну та визначити її значення за допомогою функцій
  remember { *mutableStateOf*(currentIndex) }
- Функція <T> *mutableStateOf*(value:T, policy: SnapshotMutationPolicy<T> = structuralEqualityPolicy()): MutableState<T> створює стан у межах Composable-функції, який контролює значення параметра value і у разі його зміни надає сигнал Jetpack Compose для рекомпозиції Composable-функцій, які читають цей стан.
- Функція <T> remember(crossinline calculation: DisallowComposableCalls (() -> T)):T зберігає раніше створене значення стану при рекомпозиціях.

Поняття стану

- Користувачі спілкуються із Android застосунками за допомогою **подій** (клацання, перетягування тощо).
- У відповідь на подію програма (у обробнику події) реалізує **логіку,** яка, зокрема, повинна **змінити поточний стан**.
- Зміна стану відслідковується компонентами Compose, відповідальними за UI, які **оновлюють інтерфейс,** використовуючи новий стан.



Обробка стану з односпрямованим потоком даних

- **Односпрямований потік даних (Unidirectional Data Flow (UDF))** - це підхід до розробки додатків, згідно з яким стан, що зберігається в Composable функції, не повинен безпосередньо змінюватися в жодній з дочірніх Composable функцій.
- Ключова концепція тут полягає в тому, що **стан тече вниз**, а **події течуть вгору**, як показано на зображенні вище.
- Інша ключова концепція полягає в тому, що **UI спостерігає за станом**. Щоразу, коли з'являється новий стан, інтерфейс користувача відображає його (у Compose - автоматично).
- Розглянемо приклад ієрархії Composable функцій, де батьківська функція UniDirectionalDataFlow містить стан UI компонента Checkbox (що також є Composable функцією), а сам компонент розміщений у дочірній функції CheckBoxItem.

Обробка стану з односпрямованим потоком даних - неправильна організація обробки

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

import ...

@Composable
fun UniDirectionalDataFlow(
    innerPadding: PaddingValues
) {
    val isCheckedState = remember { mutableStateOf(false) }

    CheckBoxItem ()
}

@Composable
fun CheckBoxItem(
    isCheckedState: MutableState<Boolean> = mutableStateOf(false),
) {
    Checkbox(
        checked = isCheckedState.value,
        onCheckedChange = { newValue ->
            isCheckedState.value = newValue
        }
    )
}
```

Батьківська функція

Батьківська функція містить стан UI-компонента

Виклик дочірньої функції

Дочірня функція

ПРОБЛЕМА: CheckBoxItem має параметр MutableState за замовчуванням, тому вона може створювати та володіти власним станом (що робить його неконтрольованим).

ПРОБЛЕМА: CheckBoxItem отримує стан сама від себе

ПРОБЛЕМА: CheckBoxItem змінює стан сама, так що батьківська функція про це не знає

Обробка стану з односпрямованим потоком даних - правильна організація

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

import ...

@Composable
fun UniDirectionalDataFlow(
    innerPadding: PaddingValues
) {
    val isCheckedState = remember { mutableStateOf(false) }

    val onCheckedChange = { newValue: Boolean ->
        isCheckedState.value = newValue
    }

    CheckBoxItem (
        isCheckedState = isCheckedState,
        onCheckedChange = onCheckedChange
    )
}
...
```

Батьківська функція

Батьківська функція містить стан UI-компонента

подія

Батьківська функція містить обробник подій

Виклик дочірньої функції з передачею їй стану

Дочірня функція отримує стан від батьківської

Дочірня передає подію батьківській функції через callback

Обробка стану з односпрямованим потоком даних - правильна організація

```
...
@Composable
fun CheckBoxItem(
    isCheckedState: MutableState<Boolean>,
    onCheckedChange: (Boolean) -> Unit
) {
    Checkbox(
        checked = isCheckedState.value,
        onCheckedChange = onCheckedChange
    )
}
```

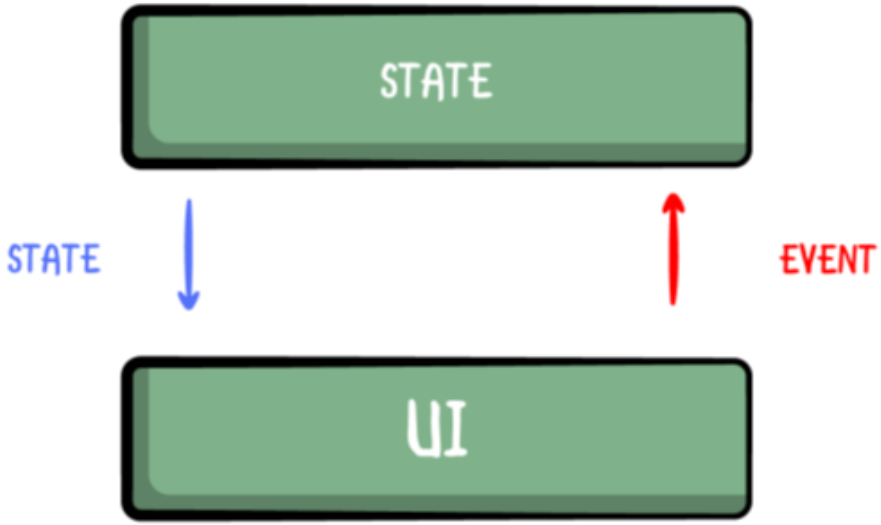
Дочірня функція

Дочірня функція отримує стан від батьківської

Дочірня передає подію батьківській функції через callback

компонент отримує стан від дочірньої функції

Компонент передає подію дочірній функції через виклик callback

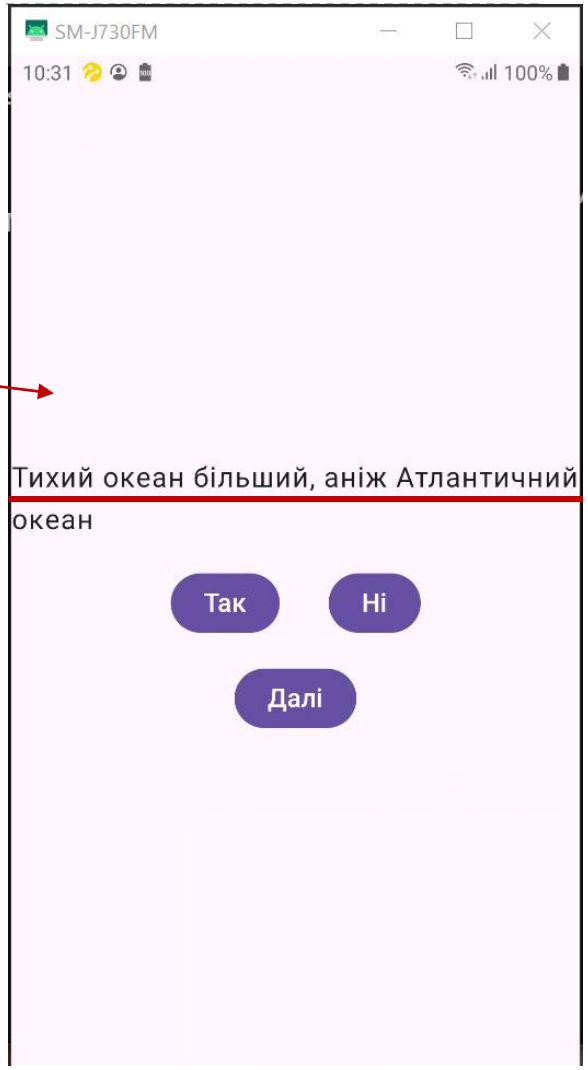


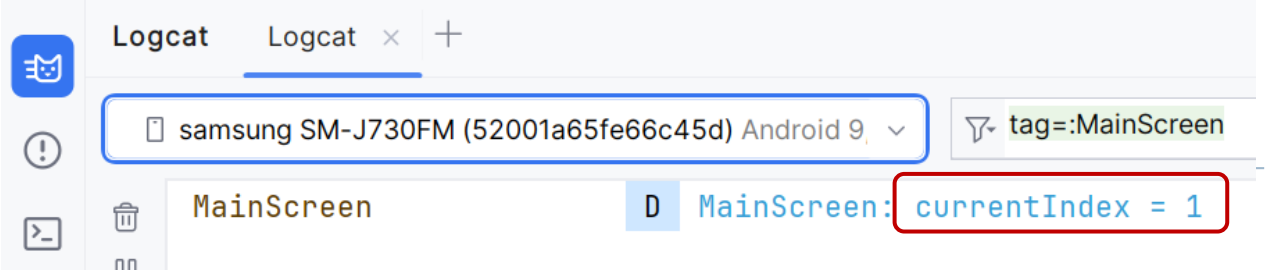
Робота зі станом

```
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),
            modifier = Modifier.padding(innerPadding)
        )

...

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

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

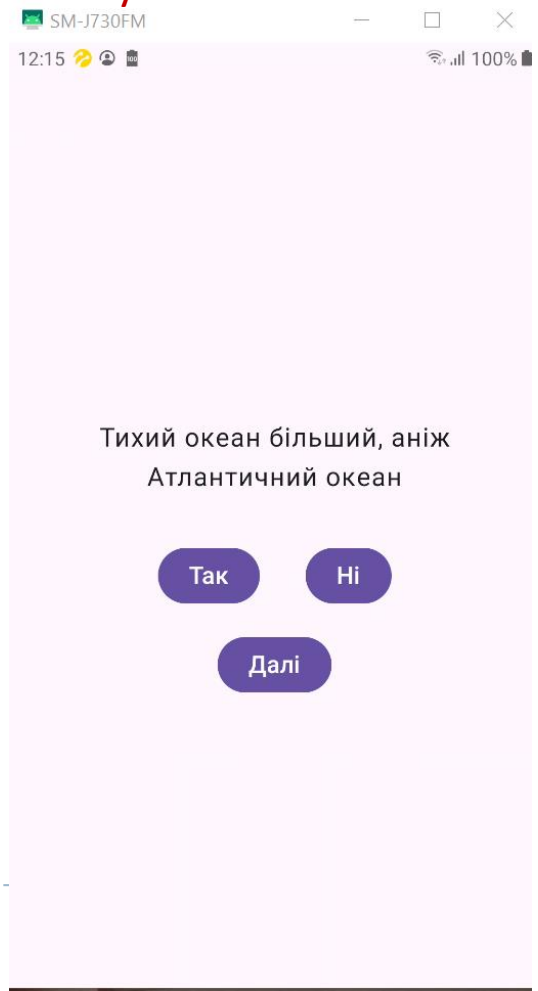
```
val currentIndexState = remember {mutableIntStateOf(currentIndex) }

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

...

Робота зі станом

```
...
    Text(
        text = stringResource(id = questionBank[currentIndexState.intValue].textResId),
        textAlign = TextAlign.Center,
        modifier = Modifier.padding(innerPadding)
            .padding(16.dp)
    )

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

Вирівнювання тексту



Робота зі станом

- Часто стан може зберігати більш, аніж одну змінну, у такому разі доцільно розробити клас даних для стану (клас повинен бути незмінним, тобто усі його властивості повинні бути оголошені як val):

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

Зараз у класі стану для спрощення тільки одна змінна

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

Робота зі станом

- Мова Kotlin має механізми, що дозволяють спрощувати код:
  - деструктуризація (Destructuring Declarations);
  - делегування властивостей (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

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

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

- У коді можна використовувати її без явного сеттера:

```
Text(
    text = stringResource(id = questionBank[mainScreenState.currentIndex].textResId),
...
Button(onClick = {
        mainScreenState = mainScreenState.copy(
            currentIndex = (mainScreenState.currentIndex + 1) % questionBank.size
        )
        setMainScreenState(newState)
        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 зберігає тільки оду змінну стандартного типу (число або рядок), функція працювати буде, оскільки вона знає як серіалізувати такі типи:

```
var currentIndexState by rememberSaveable { mutableIntStateOf(0) }
Column(
...
) {
    Text(
        // Use delegate getter for state access
        text = stringResource(id = questionBank[currentIndexState].textResId),
        ...
    )
    ...
    Button(onClick = {
        // Use delegate getter and setter for state access
        currentIndexState = (currentIndexState + 1) % questionBank.size
        // Use delegate getter for state access
        Log.d(TAG, "MainScreen: currentIndex = $currentIndexState")
    }) {
        Text(stringResource(id = R.string.next_button))        }    }}
```

Робота зі станом інтерфейса - поворот екрану

- У разі, коли стан зберігає тільки оду змінну стандартного типу (число або рядок), функція працювати буде, оскільки вона знає як серіалізувати такі типи:

```
var currentIndexState by rememberSaveable { mutableIntStateOf(0) }
Column(
...
) {
    Text(
        // Use delegate getter for state access
        text = stringResource(id = questionBank[currentIndexState].textResId),
        ...
    )
    ...
    Button(onClick = {
        // Use delegate getter and setter for state access
        currentIndexState = (currentIndexState + 1) % questionBank.size
        // Use delegate getter for state access
        Log.d(TAG, "MainScreen: currentIndex = $currentIndexState")
    }) {
        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 об1єкту цього класу в Bundle::

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

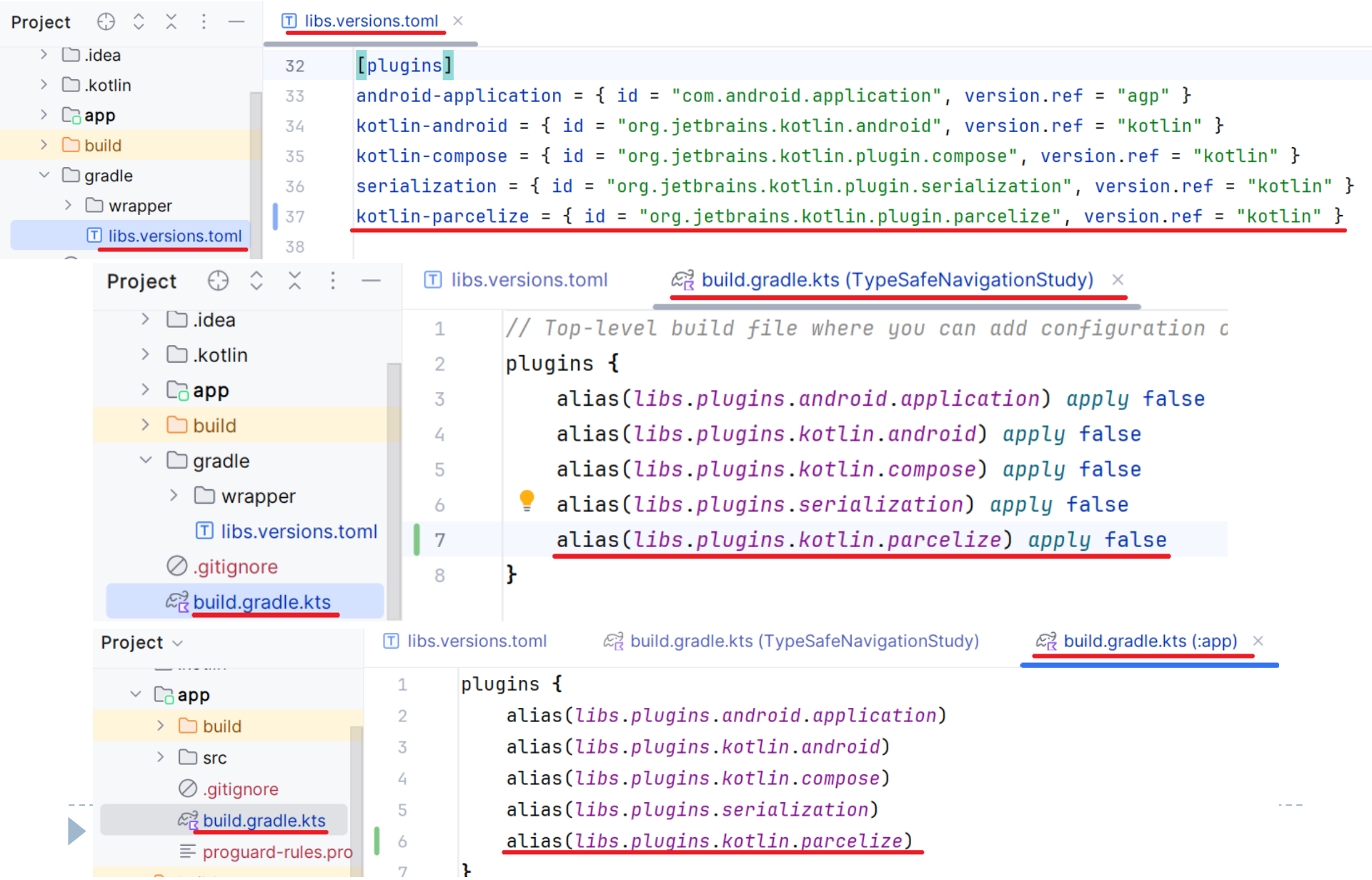
```
@Composable
fun MainScreen(
    innerPadding: PaddingValues,
) {
...
    var mainScreenState by rememberSaveable { mutableStateOf(MainScreenState()) }
...
        Text(
            // Use delegate getter for state access
            text = stringResource(id = questionBank[mainScreenState.currentIndex].textResId),
...
```

Робота зі станом інтерфейса - поворот екрану

```
...
        Text(
            // Use delegate getter for state access
            text = stringResource(id = questionBank[mainScreenState.currentIndex].textResId),
...
        Button(onClick = {
            // Use delegate getter and setter for state access
            mainScreenState =
                mainScreenState.copy((mainScreenState.currentIndex + 1) % questionBank.size)
            // Use delegate getter for state access
            Log.d(TAG, "MainScreen: currentIndex = ${mainScreenState.currentIndex}")
        }) {
            Text(stringResource(id = R.string.next_button))
        }
    }
}
```

- 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 Parcelize.

Add plugin org.jetbrains.kotlin.plugin.parcelize

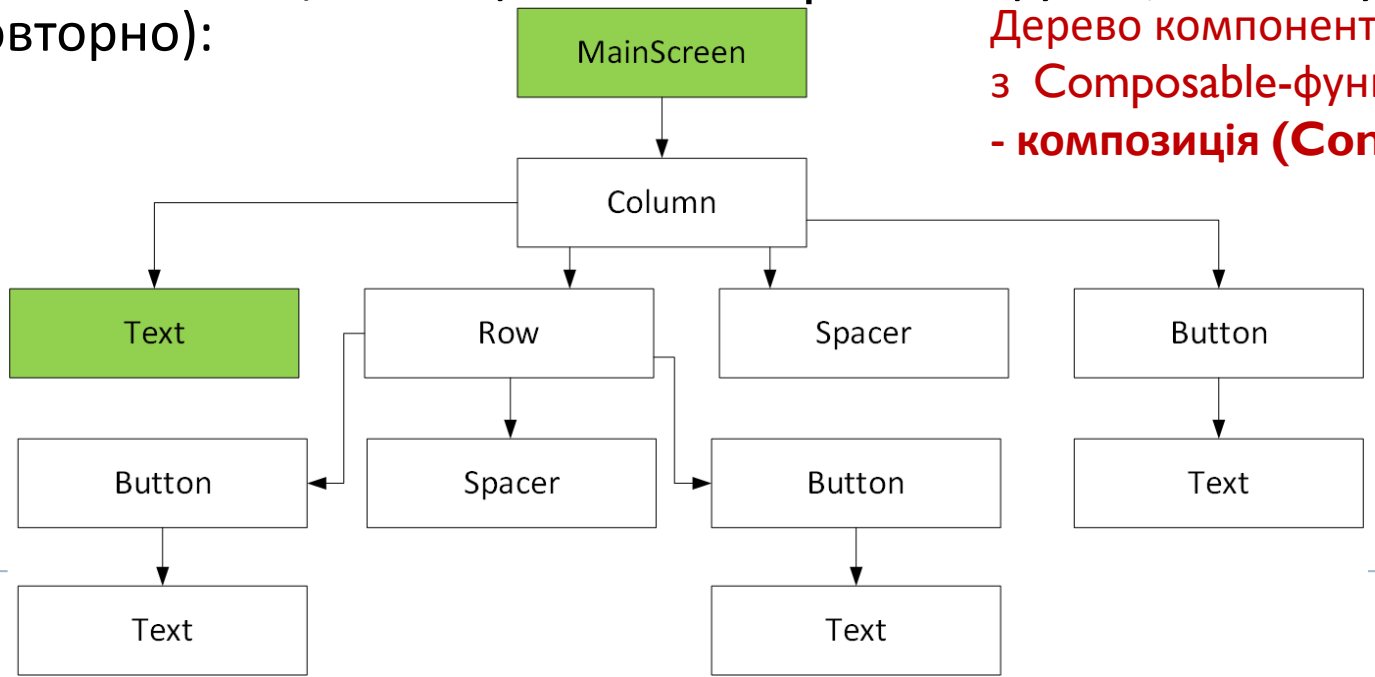


Робота зі станом інтерфейса - поворот екрану

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

@Composable
fun MainScreen(
    innerPadding: PaddingValues,
) {
    ...
    var mainScreenState by rememberSaveable { mutableStateOf(MainScreenState()) }
    Column(
        ...
    ) {
        Text(
            text = stringResource(id = questionBank[mainScreenState.currentIndex].textResId),
            ... }
        Button(onClick = {
            mainScreenState =
                mainScreenState.copy((mainScreenState.currentIndex + 1) % questionBank.size)
            Log.d(TAG, "MainScreen: currentIndex = ${mainScreenState.currentIndex}")
        }) ...
```

Композиція у Jetpack Compose

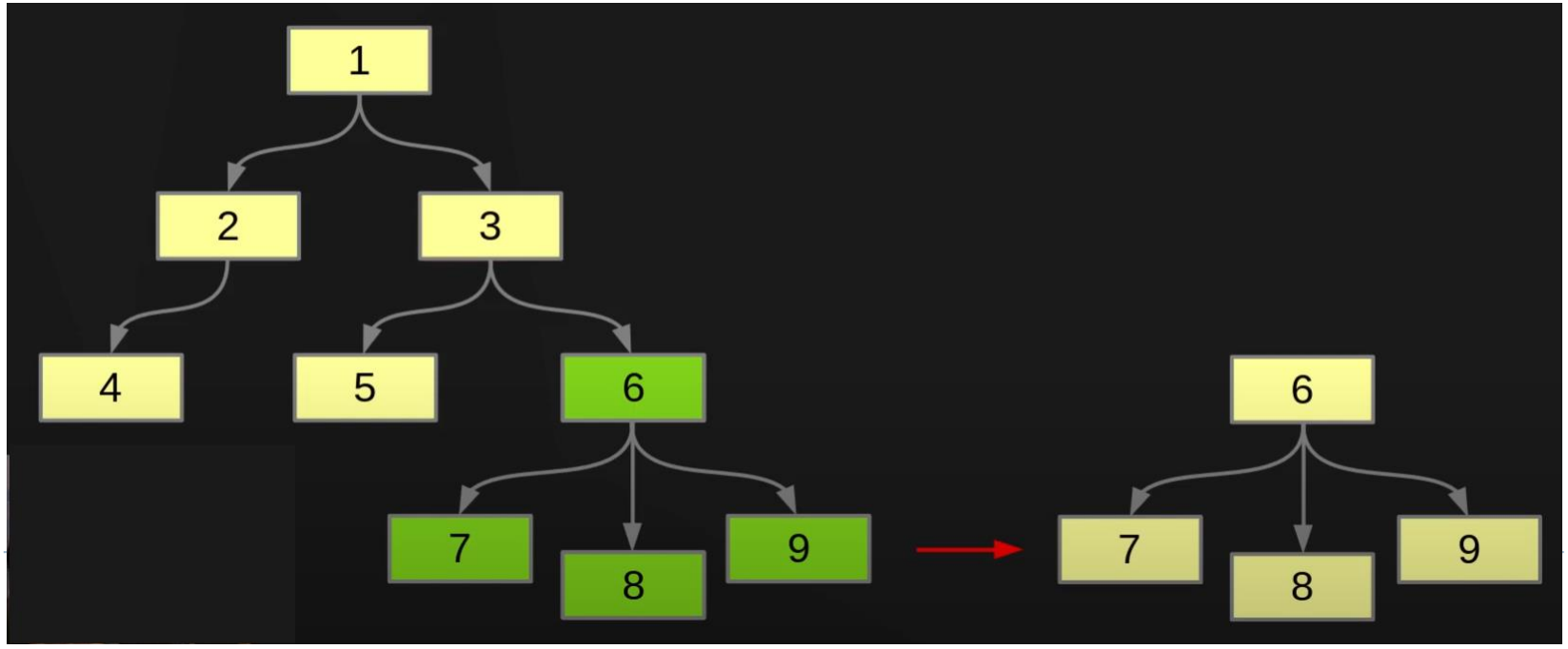
- **Композиція (Composition)** - це деревовидна структура інтерфейсу застосунку, побудована Composable-функціями.
- Оскільки компоненти інтерфейсу, створені Composable-функціями є незмінними, для їх зміни/оновлення необхідно використовувати стан.
- Зміна стану запускає процес **рекомпозиції**, який оновлює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



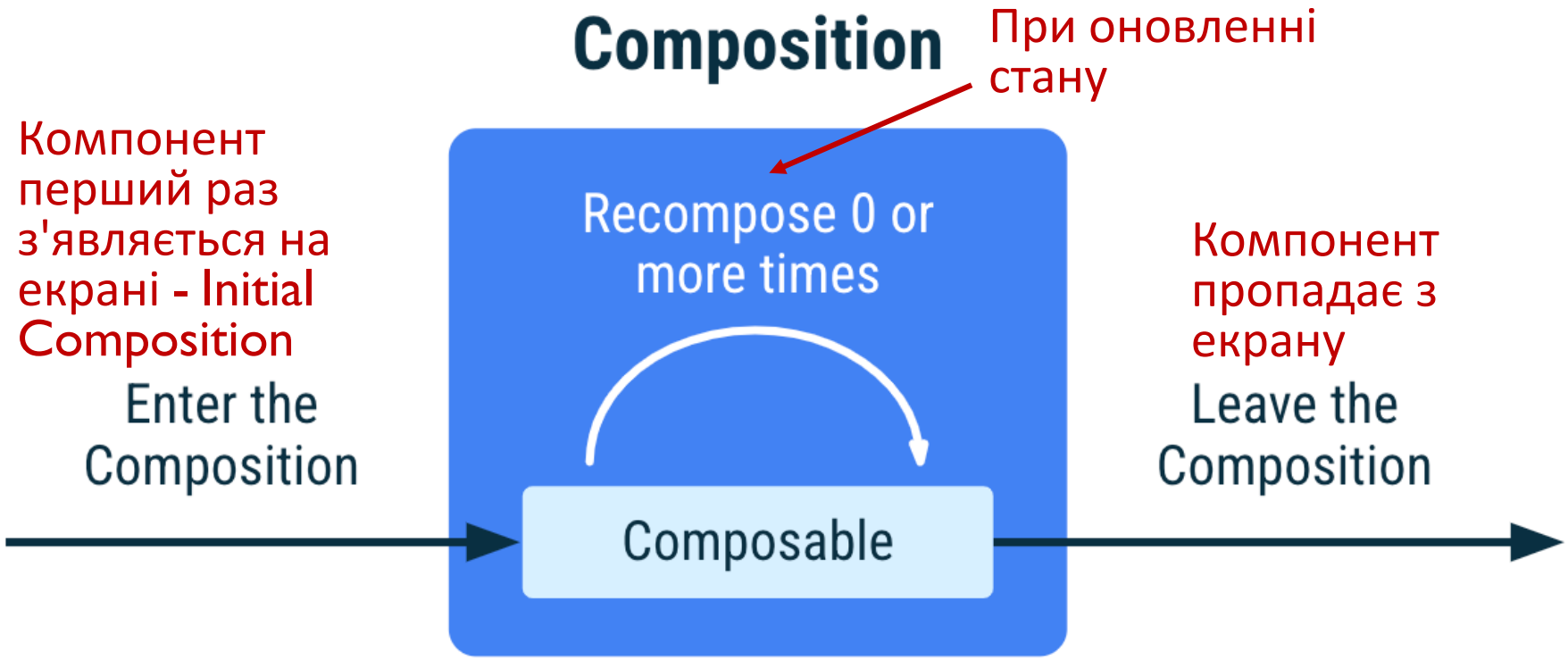
Дерево компонентів
з Composable-функцій
- композиція (Composition)

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
}
```

LifeHack - використання функції remember для логування

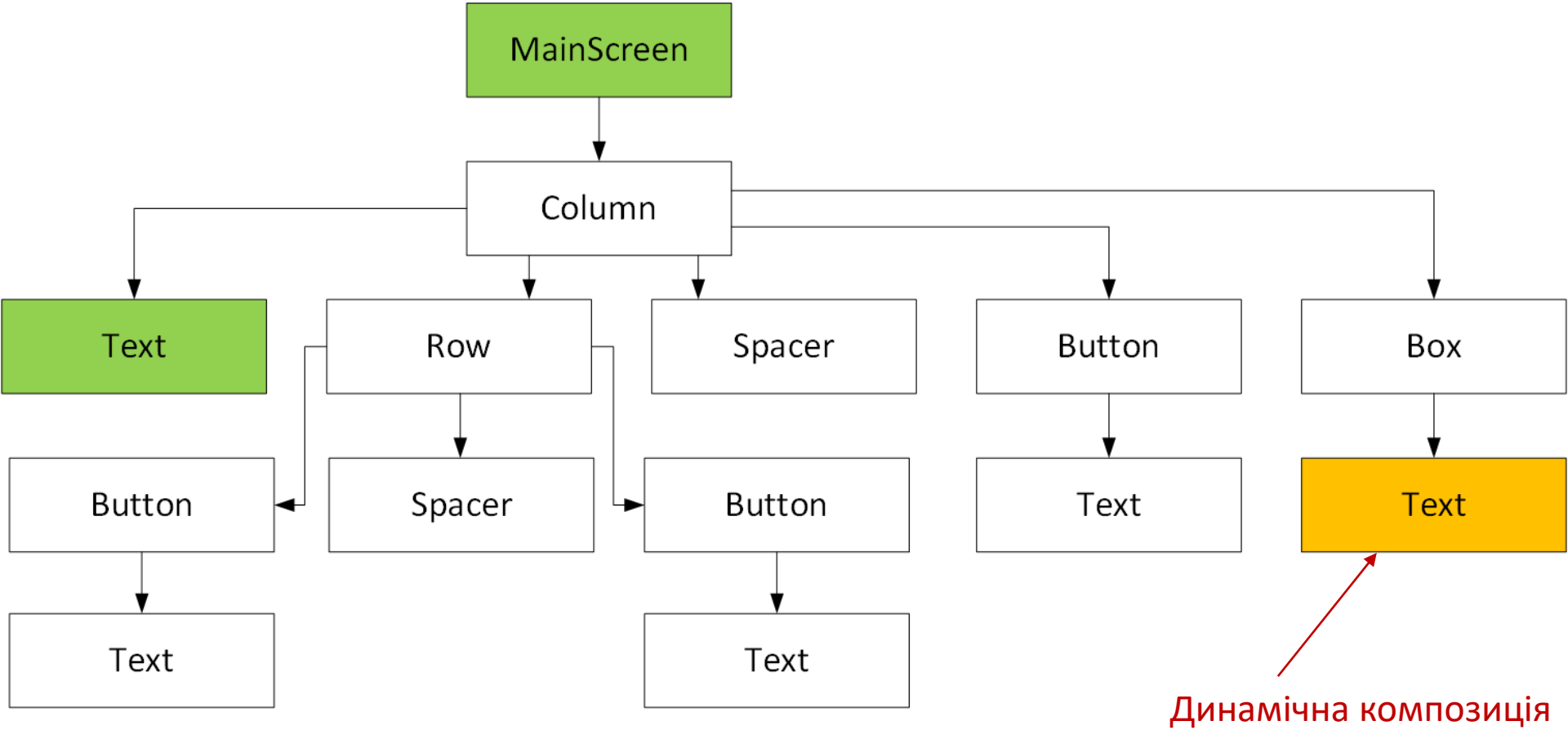
Робота зі станом інтерфейса -динамічний компонент

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

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

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"
            )
        }
    }
}
```

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()) }
logCompositionLifecycle("MainScreen")
val context = LocalContext.current
Column(
    modifier = Modifier
        .fillMaxSize()
        .padding(innerPadding),
    verticalArrangement = Arrangement.Center,
    horizontalAlignment = Alignment.CenterHorizontally
) {
    Text(
        // Use delegate getter for state access
        text = stringResource(id = questionBank[mainScreenState.currentIndex].textResId),
        textAlign = TextAlign.Center,
        modifier = Modifier
            .padding(innerPadding)
            .padding(16.dp)
    ) ...
```

Реалізація обробки подій відповідей на питання

...

```
Row(
    modifier = Modifier.size(height = 90.dp, width = 200.dp),
    horizontalArrangement = Arrangement.SpaceEvenly,
    verticalAlignment = Alignment.CenterVertically
) {
    Button(
        onClick = {
            showToast(context,
                checkAnswer(true, questionBank[mainScreenState.currentIndex])
            )
        }
    ) {
        Text(stringResource(id = R.string.true_button))
    }
    Spacer(modifier = Modifier.width(16.dp))
    Button(
        onClick = {
            showToast(context,
                checkAnswer(false, questionBank[mainScreenState.currentIndex])
            ) ...
```

Реалізація обробки подій відповідей на питання

```
...
        }) {
        Text(stringResource(id = R.string.false_button))
    }
}
Spacer(modifier = Modifier.width(16.dp))
Button(onClick = {
    mainScreenState =
        mainScreenState.copy((mainScreenState.currentIndex + 1) % questionBank.size)
    Log.d(TAG, "MainScreen: currentIndex = ${mainScreenState.currentIndex}")
}) {
    Text(stringResource(id = R.string.next_button))
}
Box(
    modifier = Modifier.height(100.dp)
) {
    if (mainScreenState.currentIndex == questionBank.size - 1) {
        logCompositionLifecycle("LastQuestionText")
        Text(
            text = "This is the last question"
        )
    } } ...
```

Реалізація обробки подій відповідей на питання

```
...
private fun showToast(context: Context, @StringRes resId: Int) {
    Toast.makeText(context, resId, Toast.LENGTH_SHORT).show()
}

private fun checkAnswer(
    userAnswer: Boolean,
    question: Question
): Int {
    if (userAnswer == question.answer) {
        return R.string.correct_toast
    } else {
        return R.string.incorrect_toast
    }
}
```