Сучасні технології мобільного програмування

Safe-type Navigation in Compose

Слайди до лекцій

The Evolution of Navigation in Compose

- Before the introduction of type-safe navigation, Jetpack Compose developers relied heavily on string-based route definitions to manage navigation between screens. This approach required manually constructing routes, embedding parameters within strings, and parsing these strings in destination composables. While functional, this method introduced several challenges:

```
// Navigating to a profile screen with an ID
navController.navigate("profile/123")
```

- In this example, "profile/123" is a string that encodes the route and its parameter. However, this simplicity comes with several drawbacks:

Challenges with String-Based Navigation:

1. **Lack of Type Safety**: Parameters are passed as strings, requiring manual extraction and conversion. If a parameter was expected to be an Int, but a String was passed, this mismatch wouldn't be caught until runtime, potentially causing crashes.

```
// Extracting the ID from the route string
val profileId = backStackEntry.arguments?
                                   .getString("id")?.toIntOrNull() ?: 0
```

If the conversion fails (e.g., due to a non-numeric string), the app could behave unexpectedly or crash.

Challenges with String-Based Navigation:

2. **Manual String Construction**: Developers had to manually concatenate and interpolate strings to create routes, leading to errors if route formatting was incorrect.

```
// Incorrectly formatted route
val userId = "123" navController.navigate("profile/userId")
                                      // Should be "$userId"
```

Such issues were common, especially in larger apps with many parameters, making the navigation logic error-prone and harder to maintain.

Challenges with String-Based Navigation:

3. **Runtime Errors**: Since the navigation routes were constructed as strings, errors in route names or parameter types weren't caught until the app was running, making debugging difficult.

```
// Navigation failure due to typo
navController.navigate("profiles/123") // Incorrect route name
```

This error would only surface at runtime, making it harder to trace and fix issues.

Challenges with String-Based Navigation:

4. **Limited Scalability**: As apps grew in complexity, managing multiple routes and parameters through strings became cumbersome, with the risk of inconsistencies increasing. This was especially challenging when maintaining navigation across different modules or features within an app.

In summary, while string-based navigation worked, it lacked the robustness required for larger, more complex applications.
The need for a more reliable and maintainable approach led to the development of type-safe navigation.

Add dependencies

Add Library Dependency

Module 'app'

Step 1.
Use the form below to find the library to add. This form uses the repositories specified in the project's build files (Google, Maven Central)

navigation-compose

Search

Enter a search query or fully-qualified coordinates (e.g. guava* or com.google.*:guava* or com.google.guava:guava:26.0)

Group ID	Artifact Name	Repository
androidx.hilt	hilt-navigation-compose	Google
androidx.navigation	navigation-compose	Google

Versions
2.9.0-alpha01
2.8.3
2.8.2
2.8.1
2.8.0

Library: androidx.navigation:navigation-compose:2.8.3

Step 2.
Assign your dependency to a configuration by selecting one of the configurations below.
Open Documentation

implementation

OK Cancel

Add dependencies

Add Library Dependency

Module 'app'

Step 1.
Use the form below to find the library to add. This form uses the repositories specified in the project's build files (Google, Maven Central)

org.jetbrains.kotlinx:kotlinx-serialization-json

Search

Enter a search query or fully-qualified coordinates (e.g. guava* or com.google.*:guava* or com.google.guava:guava:26.0)

Group ID	Artifact Name	Repository	Versions
org.jetbrains.kotlinx	kotlinx-serialization-json	Maven Central	1.7.3

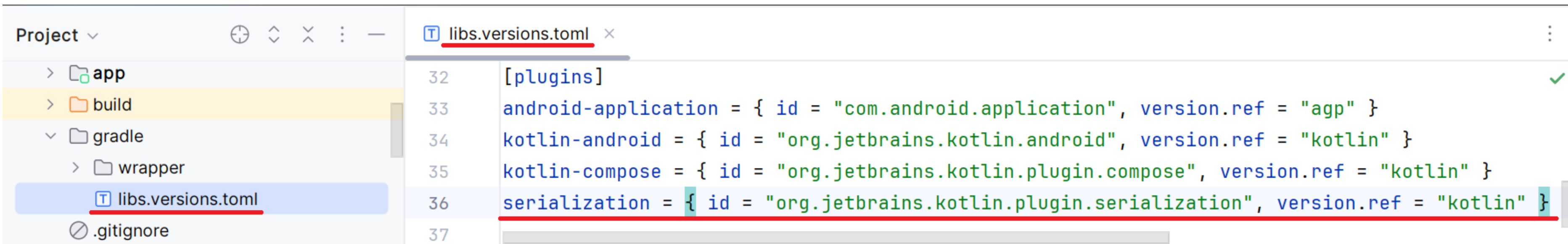
Library: org.jetbrains.kotlinx:kotlinx-serialization-json:1.7.3

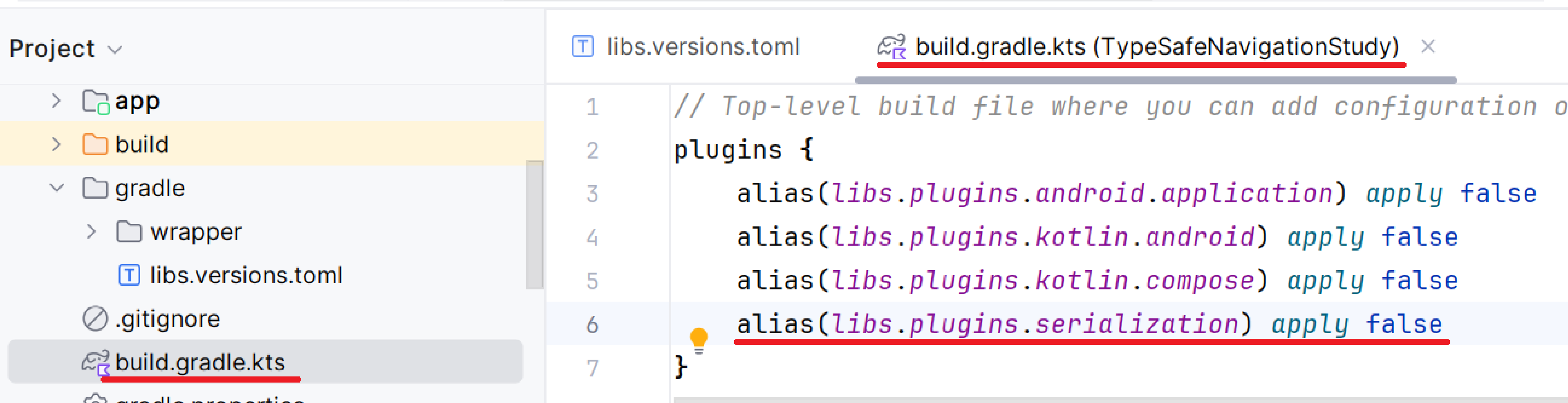
Step 2.
Assign your dependency to a configuration by selecting one of the configurations below.
Open Documentation

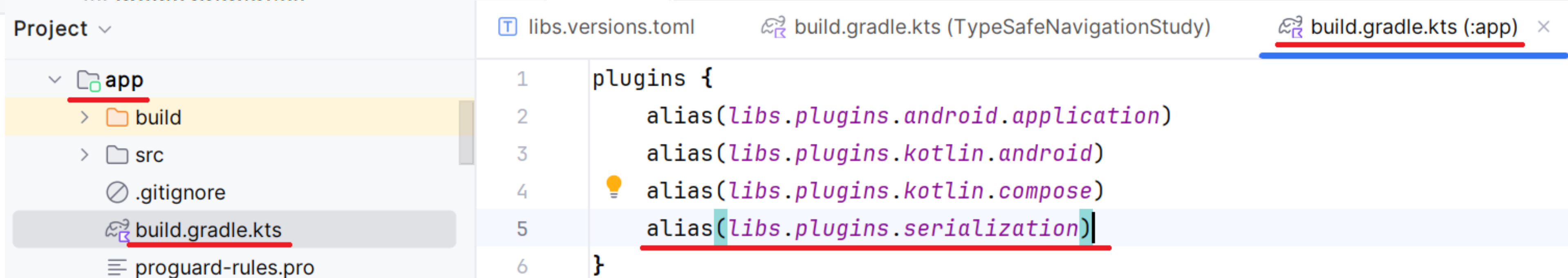
implementation

OK Cancel

Add plugin org.jetbrains.kotlin.plugin.serialization



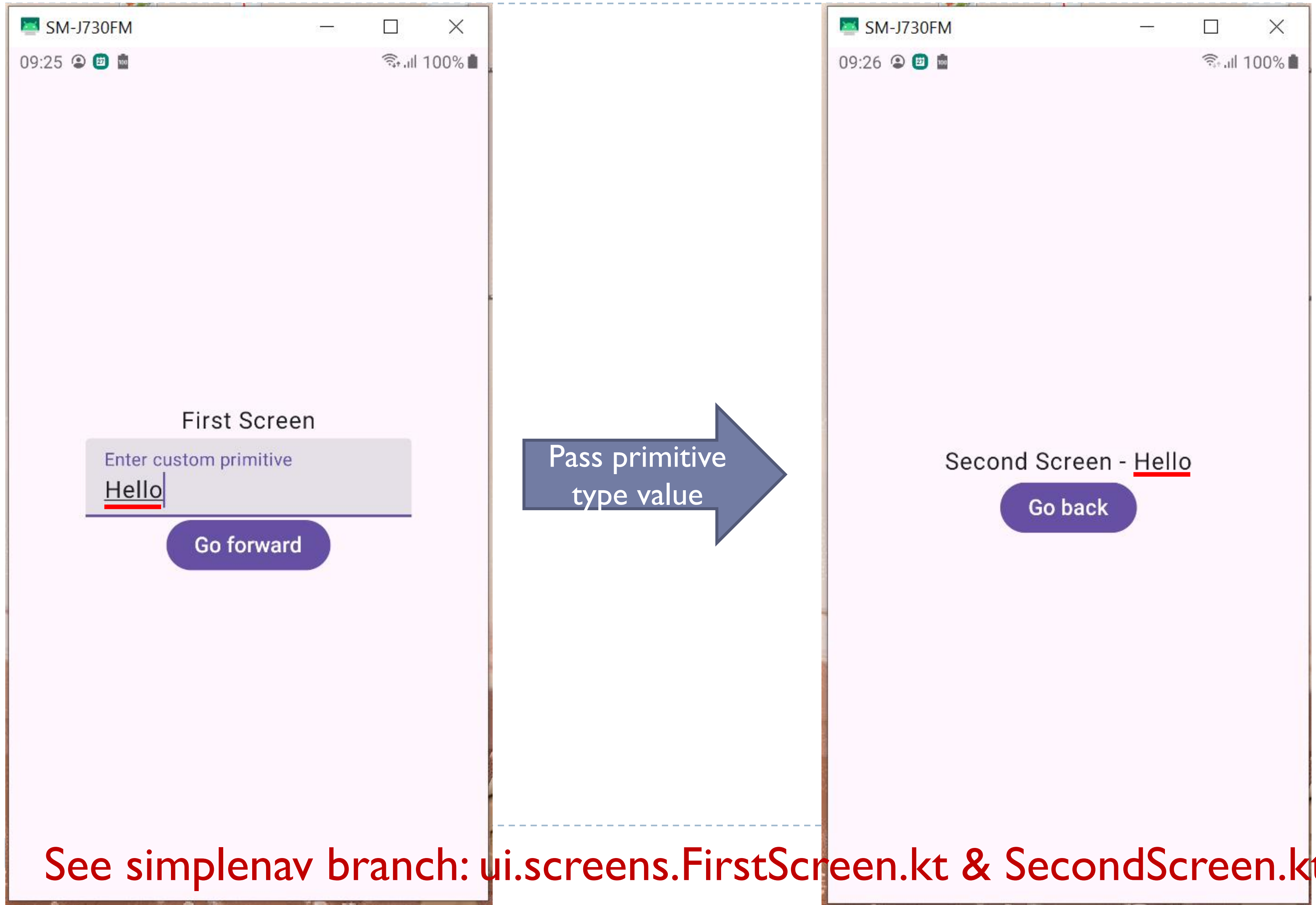




Navigation component parts

- **Navigation Graph -** a resource that collects all navigation-related data in one place. This includes all of the locations in your app, referred to as **destinations**, as well as the possible paths a user could take through your app.
- **NavHost -** a unique composable that you can include in your layout. It shows various destinations from your Navigation Graph and links the NavController with a navigation graph that specifies the composable destinations that you should be able to navigate between. As you navigate between composables, the content of the NavHost is automatically recomposed. Each composable destination in your navigation graph is associated with a route.
- **NavController** - is the central API for the Navigation component. It is stateful and keeps track of the back stack of composables that make up the screens in your app and the state of each screen.

Navigation between 2 screens - UI



See simplenav branch: ui.screens.FirstScreen.kt & SecondScreen.kt

Navigation between 2 screens - Routes

- **Kotlin Serialization** is at the core of this Type Safe Navigation, allowing developers to define destination using `@Serializable` classes.
- We need to make our classes serializable, so the arguments can be passed around.

```
// data class with custom primitive
@Serializable
data class SecondScreen(val customPrimitive: String) : Routes()
```

&

```
composable<SecondScreen> { … }
```

Instead of

```
composable("secondScreen/{customPrimitive}") { … }
```

See simplenav branch: Routes.kt

Navigation between 2 screens - Navigation Graph

1. In the new version NavHost constructors accept as startDestionation custom types, not only strings.

```
NavHost(
  navController = navController,
  /*!!!constructors accept custom types, not only strings!!!*/
  startDestination = FirstScreen,
  ...
) { ... }
```

Instead of

```
NavHost(
  navController = navController,
  startDestination = "firstScreen",
  ...
) { ... }
```

Navigation between 2 screens - Navigation Graph

2. In the new version NavHost to declare the path in the host, the composable is used a generic type, which determines, which class belongs to the destinations.

```
composable<FirstScreen> {
    ...
}
```

Instead of

```
composable("firstScreen") {
...
}
```

Navigation between 2 screens - Navigation Graph & Navigation Controller

3. In the new version NavHost to call another screen, invoke the controller as usual, but pass your data class with the values, which you need.

```
composable<FirstScreen> {
    ...
   navController.navigate(SecondScreen(customPrimitive))
   ...
}
```

Instead of

```
composable("firstScreen") {
   ...
    navController.navigate("secondScreen/${customPrimitive}");
   ...
}
```

Navigation between 2 screens - Navigation Graph

4. In the new version NavHost to get your values back, use the backStackEntry to get your value and use the value for your next screen.

```
composable<SecondScreen> { backStackEntry ->
    val route = backStackEntry.toRoute<SecondScreen>()
    val customPrimitive = route.customPrimitive
    ...
}
```

Instead of

```
composable("secondScreen/{customPrimitive}") { backStackEntry ->
    val customPrimitive = backStackEntry.arguments?
                                              .getString(" customPrimitive ")
    ...
}
```

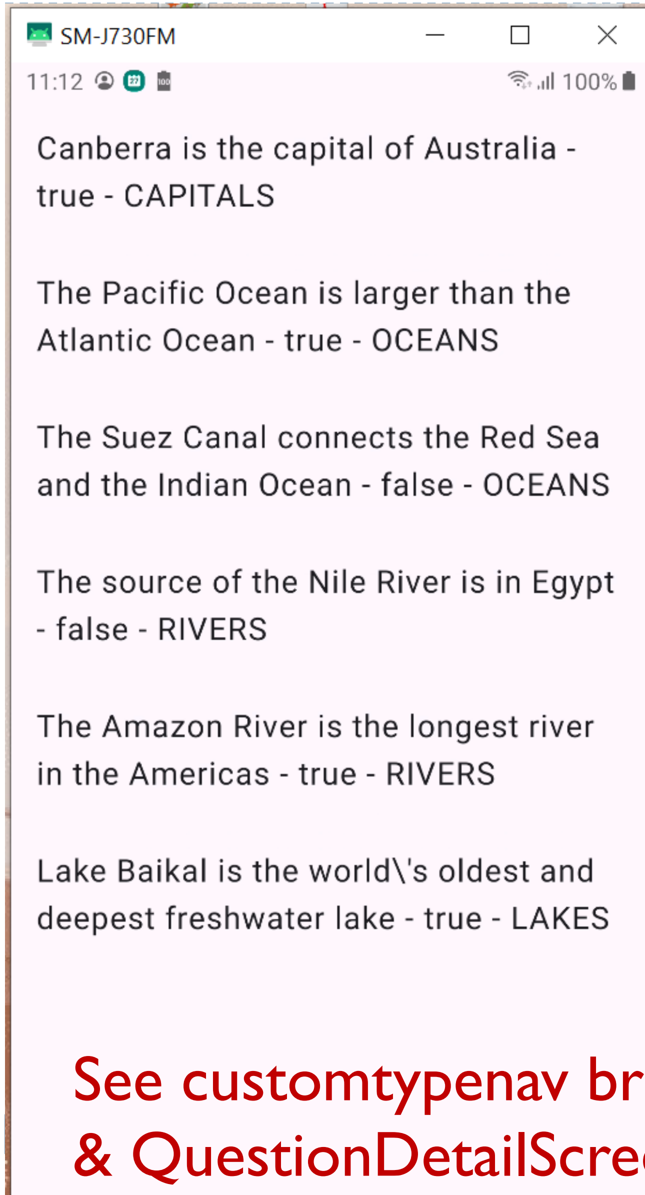
See simplenav branch: MainActivity.kt

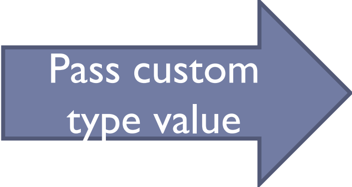
Navigation with custom type pass

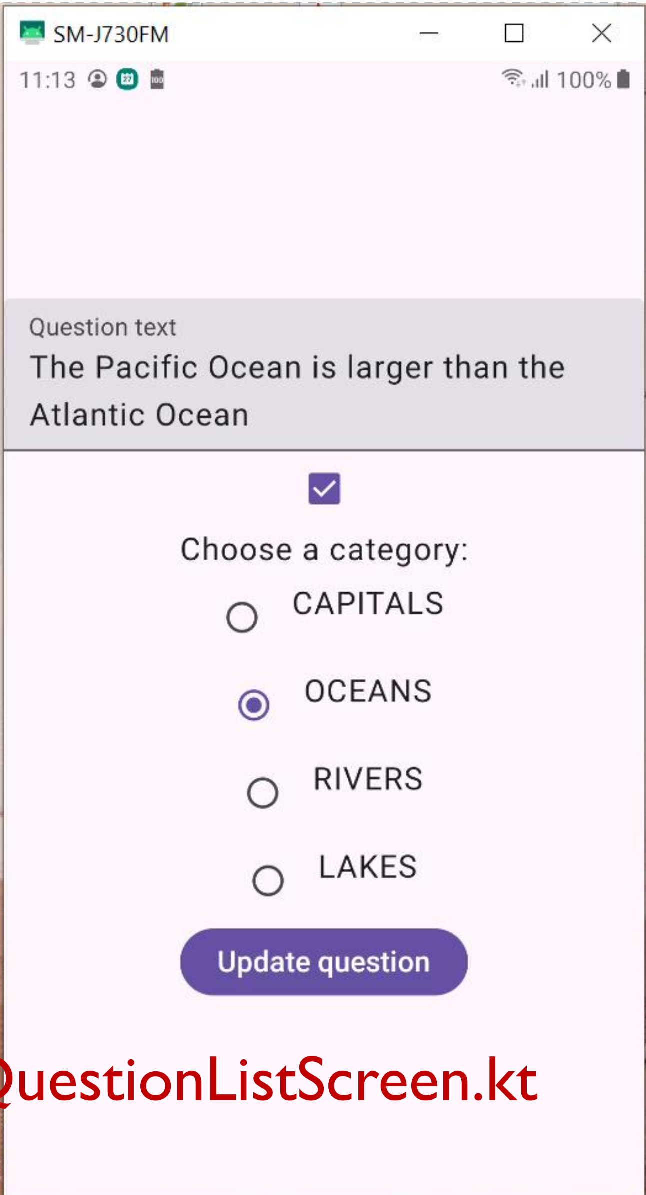
- There might be need to pass custom type instances between the screens then primitives only.
- There is a data class, instance of which will be the input for the second screen.
- We add enum element as a field of the class to demonstrate serialization-deserialization of the such values.

See customtypenav branch: model.Question.kt & model.QuestionRepository.kt

Navigation with custom type pass - UI







See customtypenav branch: ui.screens.QuestionListScreen.kt & QuestionDetailScreen.kt

Navigation with custom type pass - Routes

- **Kotlin Serialization** is at the core of this Type Safe Navigation, allowing developers to define destination using **`@Serializable`** classes.
- We need to make our classes serializable, so the arguments can be passed around.
- We use data class with custom type instance as a parameter

```
@Serializable
data class QuestionDetailRoute(val question: Question) : Routes()
```

See customtypenav branch: nav.Routes.kt

Navigation with custom type pass - NavType

- We need to define **androidx.navigation.NavType<Question>** instance with implementation of the serialization and deserialization rules of the custom type.
- Also we implements the methods for put-get the serialized custom type to Bundle. NavType <Question> instance will be used by compose internally to put it into a Bundle instance and later retrieve it.
- There are built-in NavTypes for primitive types, such as int, long, boolean, float, and strings, parcelable, and serializable classes (including Enums), as well as arrays of each supported type.

See customtypenav branch: nav.CustomNavType.kt

Navigation with custom type pass - Navigation Graph

- Now we use defined custom NavType instance with Navigation Graph as element of the Map<KType, NavType<*>> of NavGraphBuilder.composable typeMap argument.

```
NavHost(
   navController = navController,
   startDestination = Routes.QuestionListRoute,
   modifier = Modifier.padding(innerPadding)
) {
...
   composable<Routes.QuestionDetailRoute>(
      /*Custom type map for the custom type*/
      typeMap = mapOf(typeOf<Question>() to CustomNavType.questionType
   )
}
```

See customtypenav branch: MainActivity.kt

Navigation with custom type pass - Navigation Graph - WITHOUT CUSTOM NAV TYPE

```
29  class MainActivity : ComponentActivity() {
30      override fun onCreate(savedInstanceState: Bundle?) {
33          setContent {
34              TypeSafeNavigationStudyTheme {
57                          composable<Routes.QuestionDetailRoute>(
58                              /*Custom type map for the custom type*/
59  //                              typeMap = mapOf(
60  //                                  typeOf<Question>() to CustomNavType.questionType,
61  //                                  /*Enum type map for the enum type, when enum element is not a field
62  //                                    of the Question, but value of the Map<Question, Category)*/
63  ////                                typeOf<Category>() to NavType.EnumType(Category::class.java)
```

```
[ERROR:gpu_process_host.cc(982)] GPU process exited unexpectedly: exit_code=0
App trying to use insecure INPUT_FEATURE_NO_INPUT_CHANNEL flag.  Ignoring
FATAL EXCEPTION: main
Process: ua.edu.znu.typesafenavigationstudy, PID: 16714
java.lang.IllegalArgumentException: Route ua.edu.znu.typesafenavigationstudy.nav.Routes.QuestionDetailRoute could not find any NavTyp
    at androidx.navigation.serialization.RouteSerializerKt.forEachIndexedKType(RouteSerializer.kt:188)
    at androidx.navigation.serialization.RouteSerializerKt.generateRoutePattern(RouteSerializer.kt:62)
    at androidx.navigation.serialization.RouteSerializerKt.generateRoutePattern$default(RouteSerializer.kt:45)
```

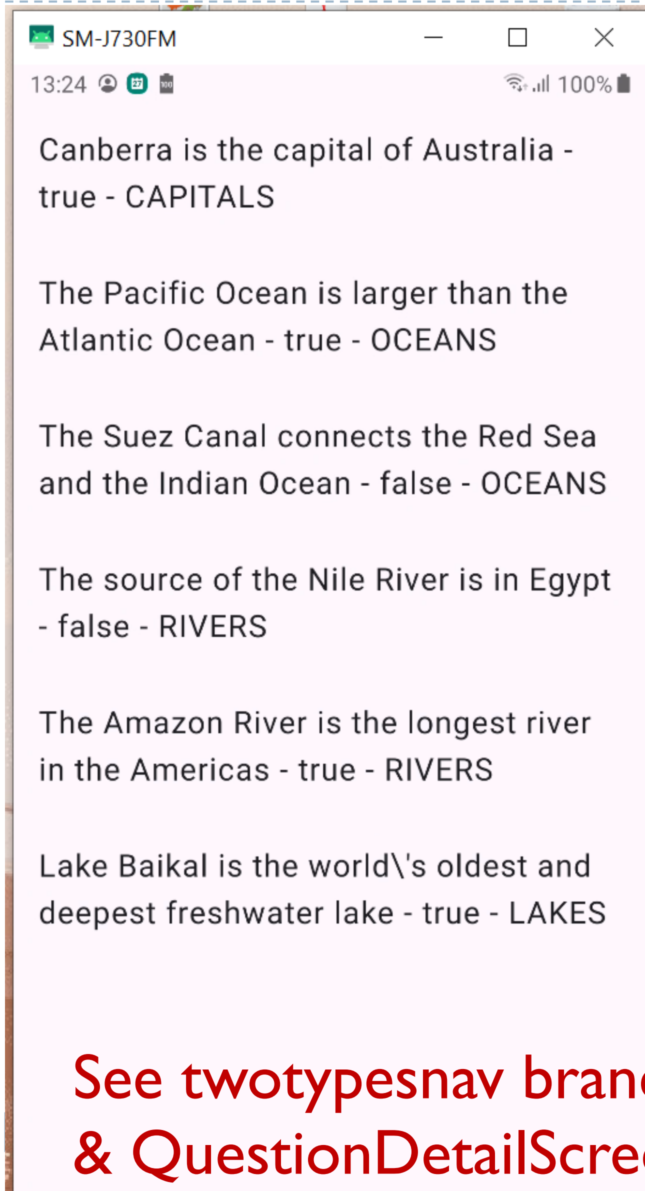
java.lang.IllegalArgumentException: Route QuestionDetailRoute could not find any NavType for argument question of type Question - typeMap received was {}

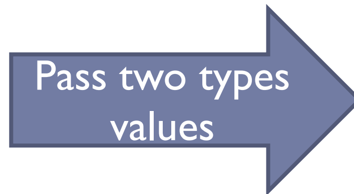
Navigation with two types pass

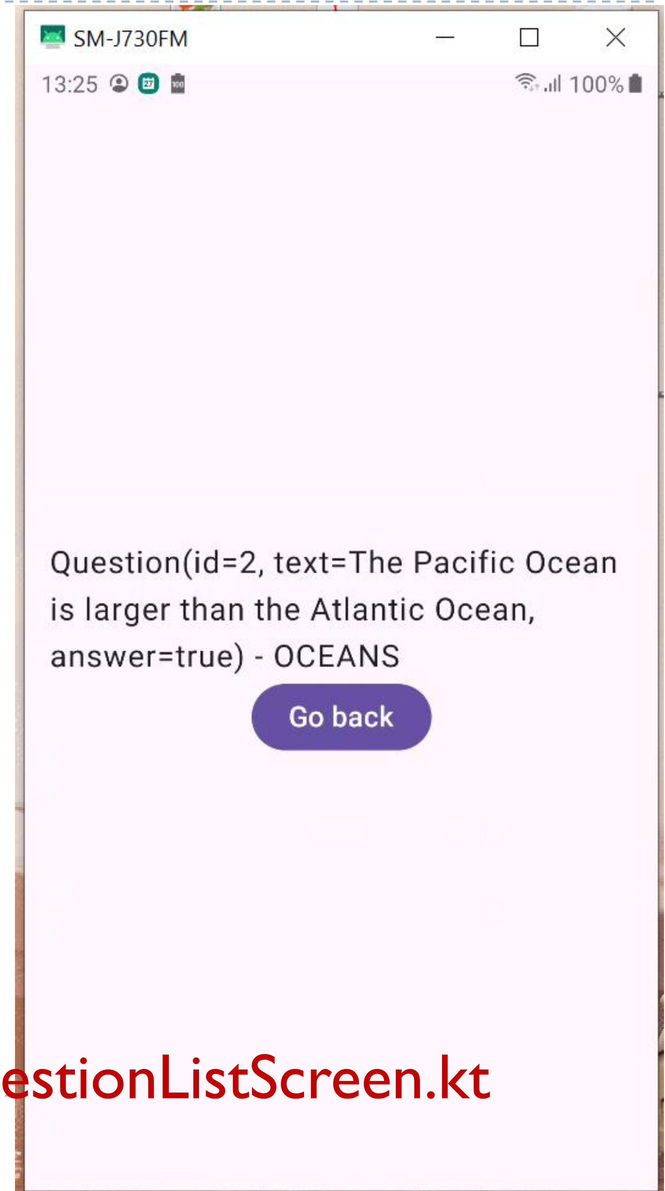
- There might be need to pass more than one type instances between the screens.
- Eq. we use Category enum as separate data structure, that will use with Question instance in the map<Question, Category>

See twotypesnav branch: model.Question.kt & model.QuestionRepository.kt

Navigation with two types pass - UI







See twotypesnav branch: ui.screens.QuestionListScreen.kt & QuestionDetailScreen.kt

Navigation with two types pass - Routes

- We need to make our classes serializable, so the arguments can be passed around.
- We use data class with two types instances as parameters.

```
@Serializable
data class QuestionDetailRoute(
    val question: Question,
    val category: Category
)
```

See twotypesnav branch: nav.Routes.kt

Navigation with two types pass - NavType

- We need to define **androidx.navigation.NavType<Question>** instance with implementation of the serialization and deserialization rules.
- The second QuestionDetailRoute argument is enum, that has standard defined NavType.

See twotypesnav branch: nav.CustomNavType.kt

Navigation with two types pass - Navigation Graph

- Now we use defined custom NavType instance with Navigation Graph as element of the Map<KType, NavType<*>> of NavGraphBuilder.composable typeMap argument.

```
NavHost(
    navController = navController,
    startDestination = Routes.QuestionListRoute,
    modifier = Modifier.padding(innerPadding)
) {
...
    composable<Routes.QuestionDetailRoute>(
        /*Custom type map for the custom types*/
        typeMap = mapOf(
            typeOf<Question>() to CustomNavType.questionType,
            typeOf<Category>() to NavType.EnumType(Category::class.java)
        )
}
```

See twotypesnav branch: MainActivity.kt

Serializable vs Parcelable object passed with Navigation

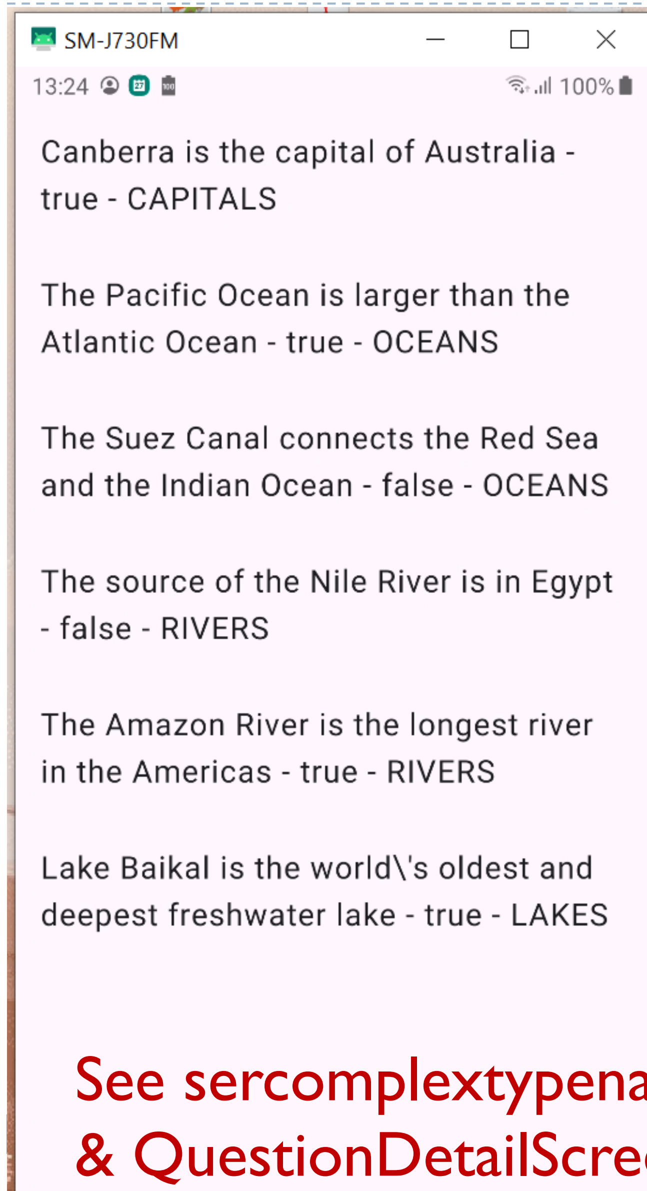
- **Serializable** is a Java interface that enables an object to be serialized, meaning that it can be converted into a byte stream and stored in a file, transmitted over a network or passed between Android components (Activities, Fragments, Bundle, Composable) as serialized string.
- **Parcelable** is an Android-specific interface that enables an object to be passed as a parameter from one Android component to another. This is a more efficient method compared to serialization, as it doesn't require the object to be converted into a byte stream (or string for Android). When an object is passed using parcelable, it is passed directly from one component to another. (eg. from Composable to Bundle and vice versa).

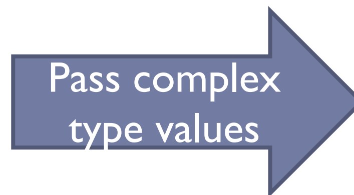
Navigation with Serializable complex type pass - data classes

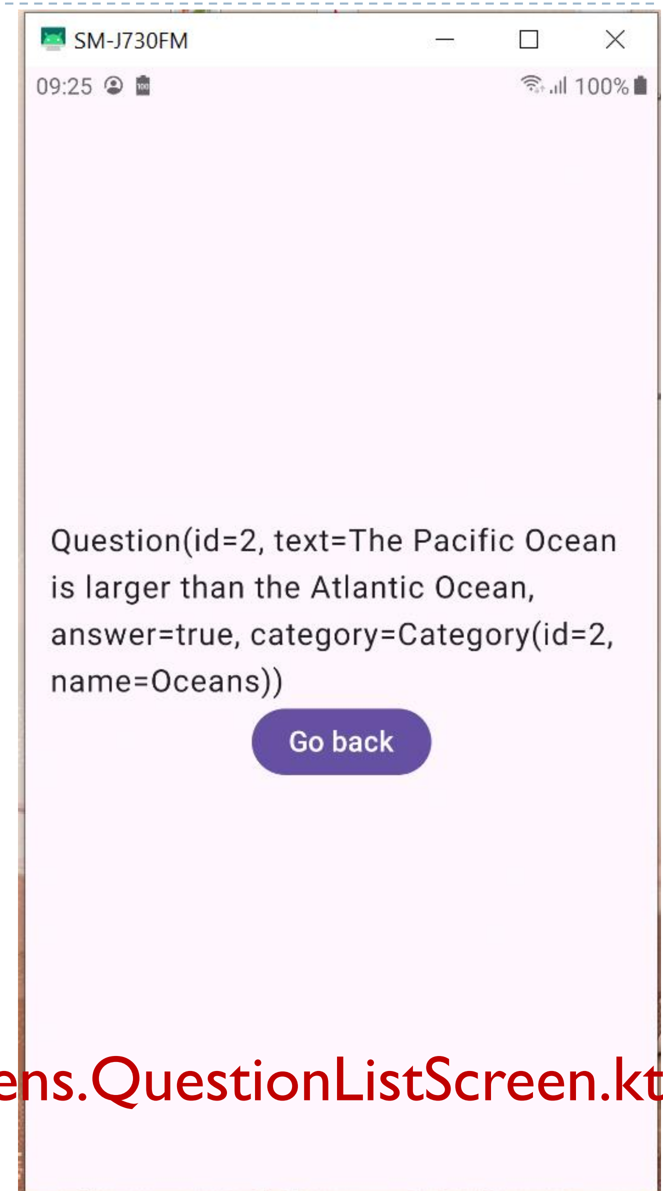
- Eg. we use Category as separate data class, that will used as a Question field.

See sercomplextypenav branch:
model.Question.kt & model.QuestionRepository.kt

Navigation with Serializable complex type pass - UI







See sercomplextypenav branch: ui.screens.QuestionListScreen.kt & QuestionDetailScreen.kt

Navigation with Serializable complex type pass - Routes

- We need to make both our classes serializable, so the Question argument can be passed around.

```
package ua.edu.znu.typesafenavigationstudy.nav

import kotlinx.serialization.Serializable
import ua.edu.znu.typesafenavigationstudy.model.Question

sealed interface Routes {

    @Serializable
    data object QuestionListRoute

    @Serializable
    data class QuestionDetailRoute(
        val question: Question
    )
}
```

See sercomplextypenav branch: nav.Routes.kt

Navigation with Serializable complex type pass - NavType

- We need to define **androidx.navigation.NavType<Question>** instance with implementation of the serialization and deserialization rules and put to and get from the Bundle methods.

Put to and get from the Bundle JSON-serialized Question instance

```
override fun get(bundle: Bundle, key: String): Question? {
    return Json.decodeFromString(bundle.getString(key) ?: return null)
}
override fun put(bundle: Bundle, key: String, value: Question) {
    bundle.putString(key, Json.encodeToString(value))
}
```

- See sercomplextypenav branch: nav.CustomNavType.kt

Navigation with Serializable complex type pass - Navigation Graph

- Now we use defined custom NavType instance with Navigation Graph as element of the Map<KType, NavType<*>> of NavGraphBuilder.composable typeMap argument.

See sercomplextypenav branch: MainActivity.kt

Navigation with Serializable complex type pass - Logcat records

Android | app | Gradle Scripts

Question.kt | CustomNavType.kt | Routes.kt | QuestionListScreen.kt | MainActivity.kt

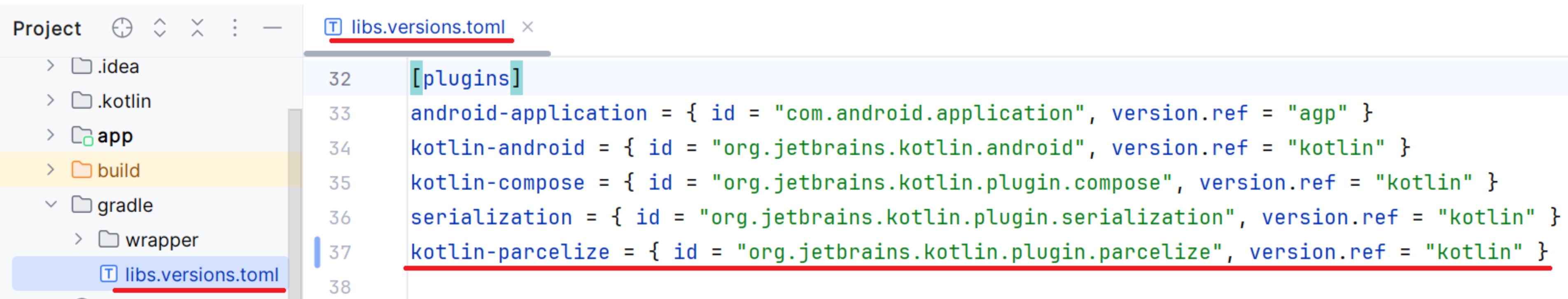
```kotlin
17   * */
18  object CustomNavType {
19      val questionType = object : NavType<Question>(
20          /* Nullable Question is not allowed */
21          isNullableAllowed = false
22      ) {
23          override fun get(bundle: Bundle, key: String): Question? {
24              Log.d(TAG, msg: "get: key = $key")
25              return Json.decodeFromString( string: bundle.getString(key) ?: return null)
26          }
27
```

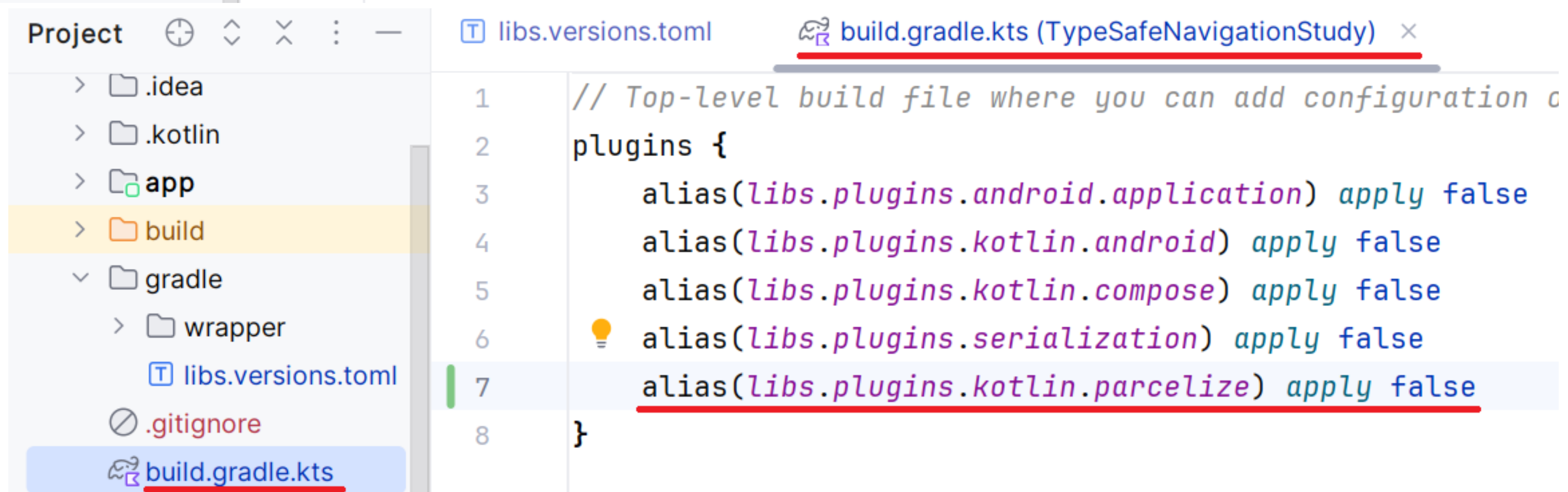
Logcat

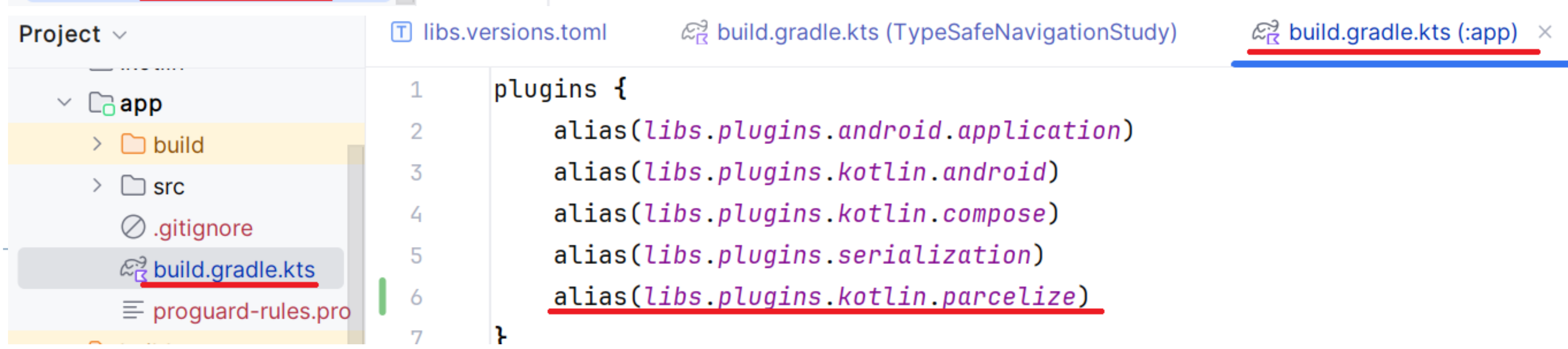
samsung SM-J730FM (52001a65fe66c45d) Android 9 | tag=:CustomNavType

```
CustomNavType  D  serializeAsValue: value = Question(id=2, text=The Pacific Ocean is larger than the Atlantic Ocean, answe
CustomNavType  D  parseValue: value = {"id":2,"text":"The Pacific Ocean is larger than the Atlantic Ocean","answer":true,"
CustomNavType  D  put: key = question, value = Question(id=2, text=The Pacific Ocean is larger than the Atlantic Ocean, an
CustomNavType  D  get: key = question
CustomNavType  D  get: key = question
CustomNavType  D  get: key = question
CustomNavType  D  get: key = question
CustomNavType  D  get: key = question
```

Add plugin org.jetbrains.kotlin.plugin.parcelize







Navigation with Parcelable complex type pass - data classes

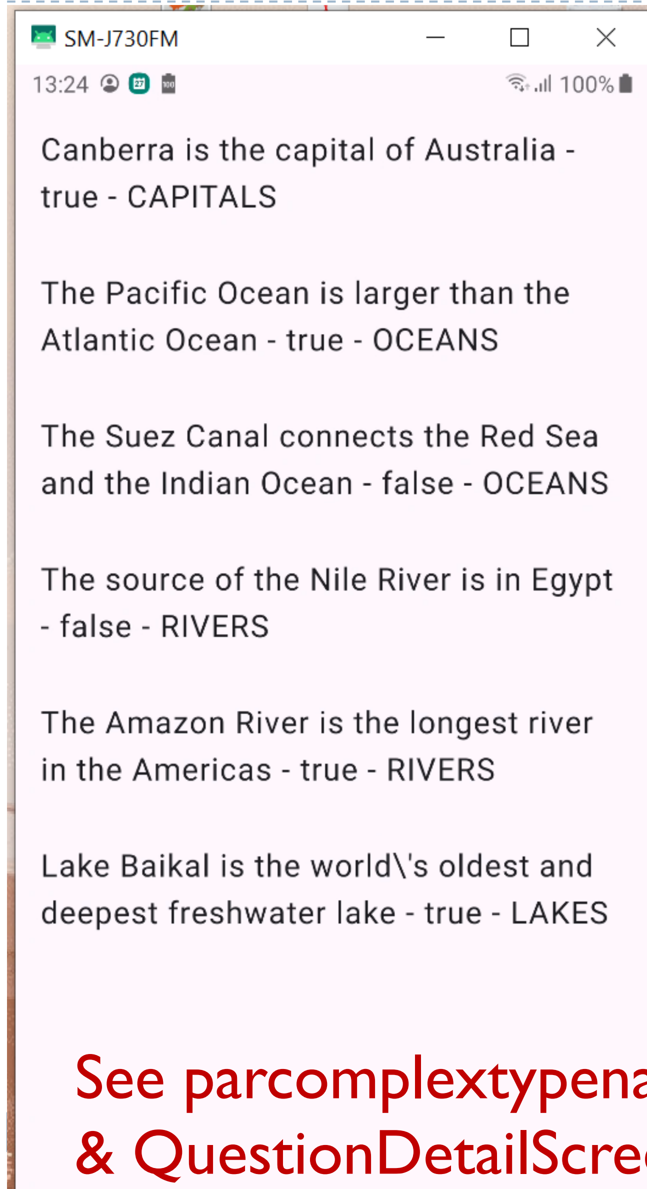
- We need use both @Serialize and @Parcelize.
- We need extends both classes from the Parcelable interface.

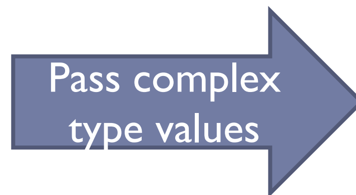
```
@Serializable
@Parcelize
data class Question(
    val id: Int,
    val text: String,
    val answer: Boolean,
    val category: Category
): Parcelable
...
```

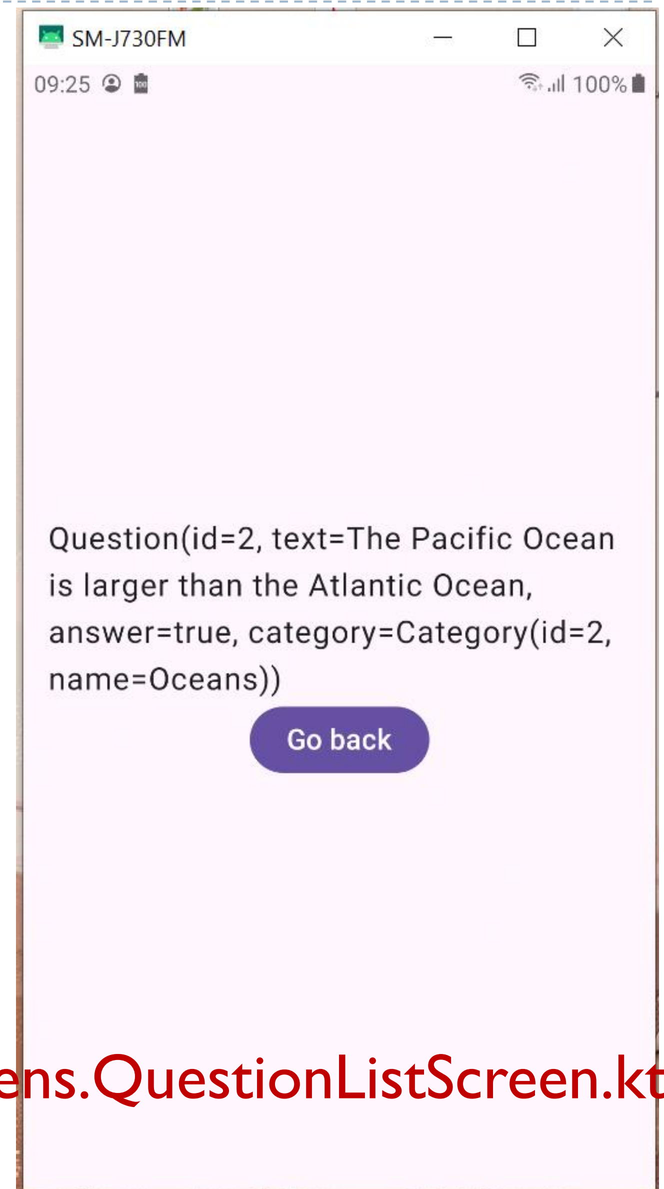
```
...
@Serializable
@Parcelize
data class Category(
    val id: Int,
    val name: String
) : Parcelable
```

See parcomplextypenav branch: model.Question.kt & model.QuestionRepository.kt

Navigation with Parcelable complex type pass - UI







See parcomplextypenav branch: ui.screens.QuestionListScreen.kt & QuestionDetailScreen.kt

Navigation with Parcelable complex type pass - Routes

- We need to make both our classes serializable, so the Question argument can be passed around.

```
package ua.edu.znu.typesafenavigationstudy.nav

import kotlinx.serialization.Serializable
import ua.edu.znu.typesafenavigationstudy.model.Question

sealed interface Routes {
    @Serializable
    data object QuestionListRoute

    @Serializable
    data class QuestionDetailRoute(
        val question: Question
    )
}
```

See parcomplextypenav branch: nav.Routes.kt

Navigation with Parcelable complex type pass - NavType

- We need to define **androidx.navigation.NavType<Question>** instance with implementation of the serialization and deserialization rules and put to and get from the Bundle methods.

Put to and get from the Bundle Question instance (without JSON serialization)

```
override fun get(bundle: Bundle, key: String): Question? {
    return if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        bundle.getParcelable(key, Question::class.java)
    } else {
        @Suppress("DEPRECATION") // for backwards compatibility
        bundle.getParcelable(key)
    }
}
override fun put(bundle: Bundle, key: String, value: Question) {
    bundle.putParcelable(key, value)
}
```

See parcomplextypenav branch: nav.CustomNavType.kt

Navigation with Parcelable complex type pass - Navigation Graph

- Now we use defined custom NavType instance with Navigation Graph as element of the Map<KType, NavType<*>> of NavGraphBuilder.composable typeMap argument.

See parcomplextypenav branch: MainActivity.kt

Navigation with Parcelable complex type pass - Logcat records

Android | Question.kt | Parcelable.java | Routes.kt | CustomNavType.kt | QuestionListScreen.kt

app
Gradle Scripts

```kotlin
18 object CustomNavType {
19     val questionType = object : NavType<Question>(
20         /* Nullable Question is not allowed */
21         isNullableAllowed = false
22     ) {
23         override fun get(bundle: Bundle, key: String): Question? {
24             Log.d(TAG, msg: "get: bundle = $bundle, key = $key")
25             return if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
26                 bundle.getParcelable(key, Question::class.java)
27             } else {
28                 @Suppress( ...names: "DEPRECATION") // for backwards compatibility
```

Logcat | Logcat

samsung SM-J730FM (52001a65fe66c45d) Android 9 | tag=:CustomNavType

```
CustomNavType  D  serializeAsValue: value = Question(id=2, text=The Pacific Ocean is larger than the Atlantic Ocean, answer
CustomNavType  D  parseValue: value = {"id":2,"text":"The Pacific Ocean is larger than the Atlantic Ocean","answer":true,"
CustomNavType  D  put: bundle = Bundle[{}], key = question, value = Question(id=2, text=The Pacific Ocean is larger than th
CustomNavType  D  get: bundle = Bundle[{question=Question(id=2, text=The Pacific Ocean is larger than the Atlantic Ocean, a
CustomNavType  D  get: bundle = Bundle[{android-support-nav:controller:deepLinkIntent=Intent { dat=android-app://androidx.r
CustomNavType  D  get: bundle = Bundle[{android-support-nav:controller:deepLinkIntent=Intent { dat=android-app://androidx.r
CustomNavType  D  get: bundle = Bundle[{android-support-nav:controller:deepLinkIntent=Intent { dat=android-app://androidx.r
CustomNavType  D  get: bundle = Bundle[{android-support-nav:controller:deepLinkIntent=Intent { dat=android-app://androidx.r
```

Navigation with Serializable vs Parcelable complex type pass

Factor	Serializable	Parcelable
Overview	Serializable is the standard Java interface for persistence.	Parcelable is the Android-specific interface for persistence.
Serialization	Objects are serialized using the Java Serialization API.	Objects are serialized using the Android Parcelable API.
Memory Usage	Serializable objects are stored in memory and can be retrieved quickly.	Parcelable objects are stored in an Android application bundle and require more time to access.
Speed	Serializable is slower than Parcelable.	Parcelable is faster than Serializable.
Size	Serializable objects are larger than Parcelable objects.	Parcelable objects are smaller than Serializable objects.
Implementation	Serializable objects are implemented by implementing the Serializable interface.	Parcelable objects are implemented by extending the Parcelable class.

Navigation with Serializable vs Parcelable complex type pass

Factor	Serializable	Parcelable
Hierarchy	Serializable supports class hierarchy.	Parcelable does not support class hierarchy.
Reflection	Serializable objects can be accessed using Java's reflection API.	Parcelable objects cannot be accessed using Java's reflection API.
Thread Safety	Serializable objects are not thread-safe	Parcelable objects are thread-safe