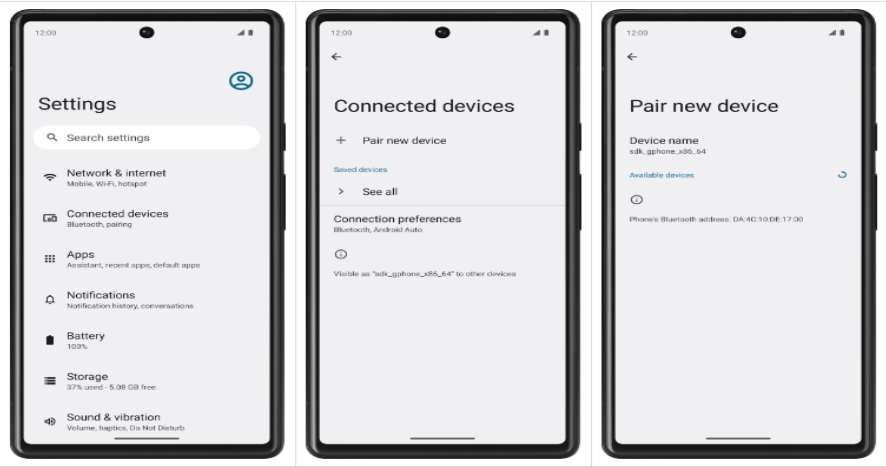
화면 전환

(Navigation)

Navigation

- 앱 내에서 다양한 화면 간의 이동을 할 수 있게 하는 핵심요소



- Navigation의 특징
  - 일관된 navigation 패턴을 제공하여 쉽게 앱 탐색을 할 수 있게 함
  - 화면 간의 이동을 추상화 하여 코드의 가독성과 유지보수성을 높임
  - Navigation 구성요소를 통해 화면 간 전환 시 라이프사이클 관리 및 상태 저장을 자동으로 처리할 수 있음

Navigation의 구성요소

- 라이브러리 추가 : androidx.navigation:navigation-compose
- **NavController** : 앱의 화면간 이동을 담당
- **NavGraph** : 이동할 컴포저블 대상을 매핑하는 역할
- **NavHost** : NavGraph의 현재 대상을 표시하는 컨테이너 역할을 하는 컴포저블

```
val navController : NavHostController = rememberNavController()

NavHost(navController=navController, startDestination = "Home"){ this: NavGraphBuilder
    composable(route="Home"){ this: AnimatedContentScope  it: NavBackStackEntry
        MainScreen1(navController)
    }
    composable(route="A"){ this: AnimatedContentScope  it: NavBackStackEntry
        ScreenA1(navController)
    }
    composable(route="B"){ this: AnimatedContentScope  it: NavBackStackEntry
        ScreenB1(navController)
    }
    composable(route="C"){ this: AnimatedContentScope  it: NavBackStackEntry
        ScreenC1(navController)
    }
}
```

주어진 람다식으로 NavGraph 생성

식별하는 키

```
navController.navigate( route: "A")
```

NavController의 navigate 함수를 통해 화면 이동

Navigation 예

```
// Define the Profile composable.
@Composable
fun Profile(onNavigateToFriendsList: () -> Unit) {
  Text("Profile")
  Button(onClick = { onNavigateToFriendsList() }) {
    Text("Go to Friends List")
  }
}

// Define the FriendsList composable.
@Composable
fun FriendsList(onNavigateToProfile: () -> Unit) {
  Text("Friends List")
  Button(onClick = { onNavigateToProfile() }) {
    Text("Go to Profile")
  }
}

// Define the MyApp composable, including the `NavController` and `NavHost`.
@Composable
fun MyApp() {
  val navController = rememberNavController()
  NavHost(navController, startDestination = "profile") {
    composable("profile") { Profile(onNavigateToFriendsList = { navController.navigate("friendslist")
    composable("friendslist") { FriendsList(onNavigateToProfile = { navController.navigate("profile")
  }
}
```

경로 문자열

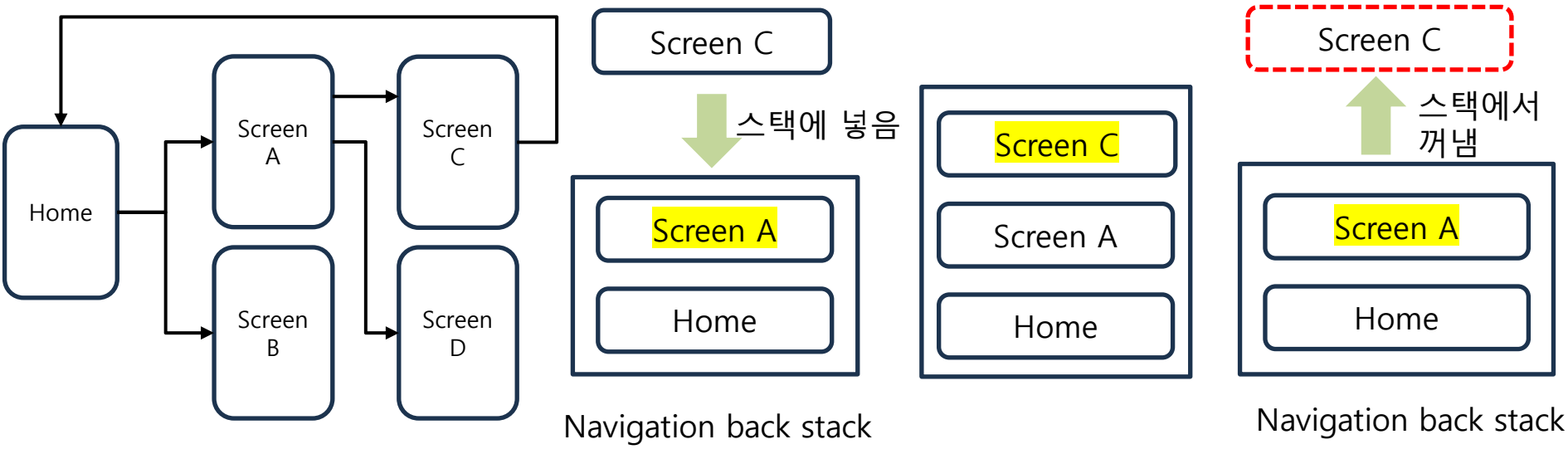
- 경로를 나타내는 객체 생성

```
sealed class Routes(val route:String){
    object Home:Routes("Home")
    object A:Routes("A")
    object B:Routes("B")
    object C:Routes("C")
}

composable(route= Routes.Home.route)
```

Navigation 경로 추적

- Navigation back stack을 이용하여 목적지에 이르는 경로 추적
  - 화면을 이동할 때 마다 화면들이 내비게이션 백 스택에 쌓임
  - 내비게이션 백 스택의 Top에 있는 화면이 현재 목적지가 됨



Navigation Back Stack 옵션

- 현재 백 스택 지우기 (popUpTo)
  - 예) Home 전까지의 컴포저블 꺼냄

```
navController.navigate("C"){
    popUpTo("Home")
}
```

- 백 스택 모두 지우기 (inclusive)
  - 예) Home을 포함한 컴포저블 꺼냄

```
navController.navigate("C"){
    popUpTo("Home") { inclusive = true }
}
```

- 백 스택 상단에 복사본 방지 (launchSingleTop)
  - 해당 목적지가 이미 스택에 있는 경우
    기존 인스턴스 재사용 없는 경우 새로 추가함

```
navController.navigate("C"){
    launchSingleTop = true
}
```

- saveState / restoreState 옵션 true
  - 백 스택 항목 상태를 자동 저장 및 복원

Log Method

- Log.v(“tag”,”message”)
  - Verbose : 개발중에만 사용하여 상세 정보 표시
- Log.i(“tag”,”message”)
  - Information : 일반 정보 표시
- Log.d(“tag”,”message”)
  - Debug : debug용 로그
- Log.w(“tag”,”message”)
  - Warning : 경고 표시
- Log.e(“tag”,”message”)
  - error : error용 로그

목적지에 인수 전달하기

- composable에 인자로 전달
  - 최소한의 정보만 인자로 전달할 것을 권고하고 있음
    - 복잡한 객체 전달은 하지 말 것을 권고하고 있음
- Route에서 값을 넘겨주기

NavGraph에서 값 받을 때

```
NavHost(startDestination = "profile/{userId}") {
    ...
    composable(
        "profile/{userId}",
        arguments = listOf(navArgument("userId") { type = NavType.StringType })
    ) {...}
}
```

모든 값은 String 타입으로 처리되므로, 타입 잘 지정할 것

```
composable("profile/{userId}") { backStackEntry ->
    Profile(navController, backStackEntry.arguments?.getString("userId"))
}
```

Navigate로 값을 넘겨줄때

```
navController.navigate("profile/user1234")
```

목적지에 인수 전달하기

- 옵션 인수 추가
  - defaultValue / nullable 인수 추가할 수 있음
  - key-value 형태로 인수 전달

```
composable(
    "profile?userId={userId}",
    arguments = listOf(navArgument("userId") { defaultValue = "user1234" })
) { backStackEntry ->
    Profile(navController, backStackEntry.arguments?.getString("userId"))
}
```

```
navController.navigate("profile?userId=user1234")
```

목적지에 인수 전달하기

- ViewModel 이용하기
  - **주의) ViewModel은 Navigation 내부에서는 공유되지 않음**

```
@Composable
fun rememberViewModelStoreOwner(): ViewModelStoreOwner {
    val context = LocalContext.current
    return remember(context) { context as ViewModelStoreOwner }
}
```

```
val LocalNavGraphViewModelStoreOwner =
    staticCompositionLocalOf<ViewModelStoreOwner> {
        error("Undefined")
    }
```

```
val navStoreOwner = rememberViewModelStoreOwner()
CompositionLocalProvider(
    LocalNavGraphViewModelStoreOwner provides navStoreOwner
) {
    NavHost(navController = navController, startDestination = Routes.Home.route) {
        composable(route = Routes.Home.route) {
            HomeScreen(navController)
        }
```

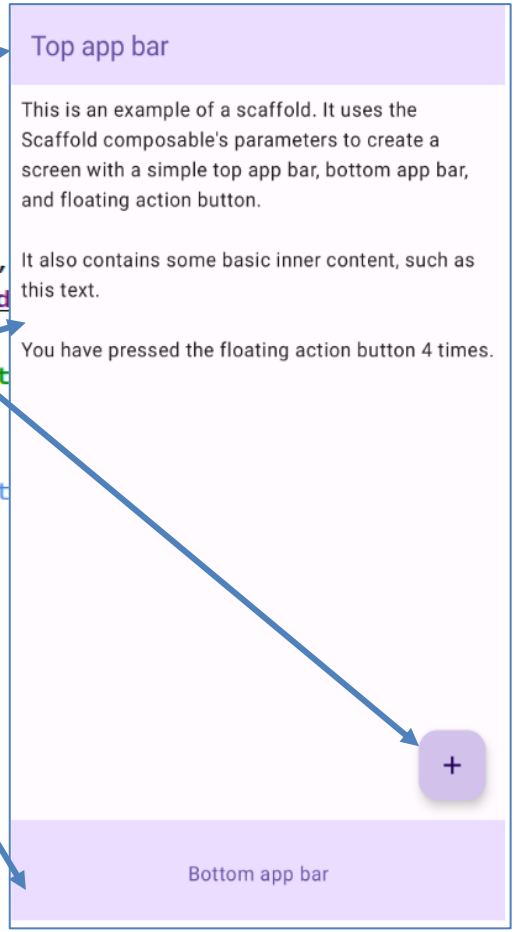
뷰모델 이용

```
val navViewModel: NavViewModel = viewModel(viewModelStoreOwner = LocalNavGraphViewModelStoreOwner.current)
```

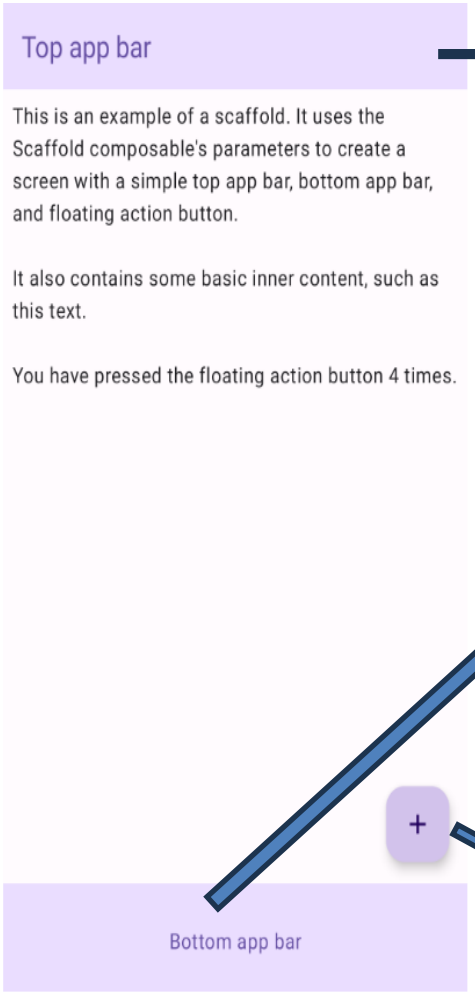
Scaffold

- 복잡한 사용자 인터페이스를 위한 표준화된 플랫폼을 제공하는 구조

```
@Composable
fun Scaffold(
    modifier: Modifier = Modifier,
    topBar: @Composable () -> Unit = {},
    bottomBar: @Composable () -> Unit = {},
    snackbarHost: @Composable () -> Unit = {},
    floatingActionButton: @Composable () -> Unit = {},
    floatingActionButtonPosition: FabPosition = FabPosition.End,
    containerColor: Color = MaterialTheme.colorScheme.background
    contentColor: Color = contentColorFor(containerColor),
    contentWindowInsets: WindowInsets = ScaffoldDefaults.content
    content: @Composable (PaddingValues) -> Unit
) {
    Surface(modifier = modifier, color = containerColor, content
        ScaffoldLayout(
            fabPosition = floatingActionButtonPosition,
            topBar = topBar,
            bottomBar = bottomBar,
            content = content,
            snackbar = snackbarHost,
            contentWindowInsets = contentWindowInsets,
            fab = floatingActionButton
        )
    }
}
```

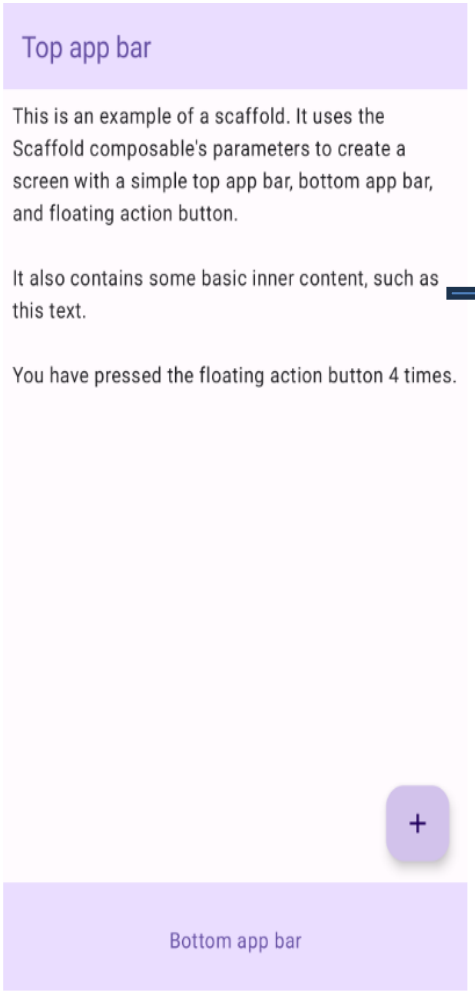


Scaffold




```
var presses : Int by remember { mutableIntStateOf( value: 0) }
Scaffold(
    topBar = {
        TopAppBar(
            colors = TopAppBarDefaults.topAppBarColors(
                containerColor = MaterialTheme.colorScheme.primaryContainer,
                titleContentColor = MaterialTheme.colorScheme.primary,
            ),
            title = {
                Text( text: "Top app bar")
            }
        )
    },
    bottomBar = {
        BottomAppBar(
            containerColor = MaterialTheme.colorScheme.primaryContainer,
            contentColor = MaterialTheme.colorScheme.primary,
        ) { this: RowScope
            Text(
                modifier = Modifier
                    .fillMaxWidth(),
                textAlign = TextAlign.Center,
                text = "Bottom app bar",
            )
        }
    },
    floatingActionButton = {
        FloatingActionButton(onClick = { presses++ }) {
            Icon(Icons.Default.Add, contentDescription = "Add")
        }
    } )
```

Scaffold




```
{ innerPadding : PaddingValues ->
    Column(
        modifier = Modifier
            .padding(innerPadding),
        verticalArrangement = Arrangement.spacedBy(16.dp),
    ) { this: ColumnScope
        Text(
            modifier = Modifier.padding(8.dp),
            text =
            """
                This is an example of a scaffold. It uses the Scaffold composable's parameters to create a s

                It also contains some basic inner content, such as this text.

                You have pressed the floating action button $presses times.
            """.trimIndent(),
        )
    }
}
```

BottomBarNavigation

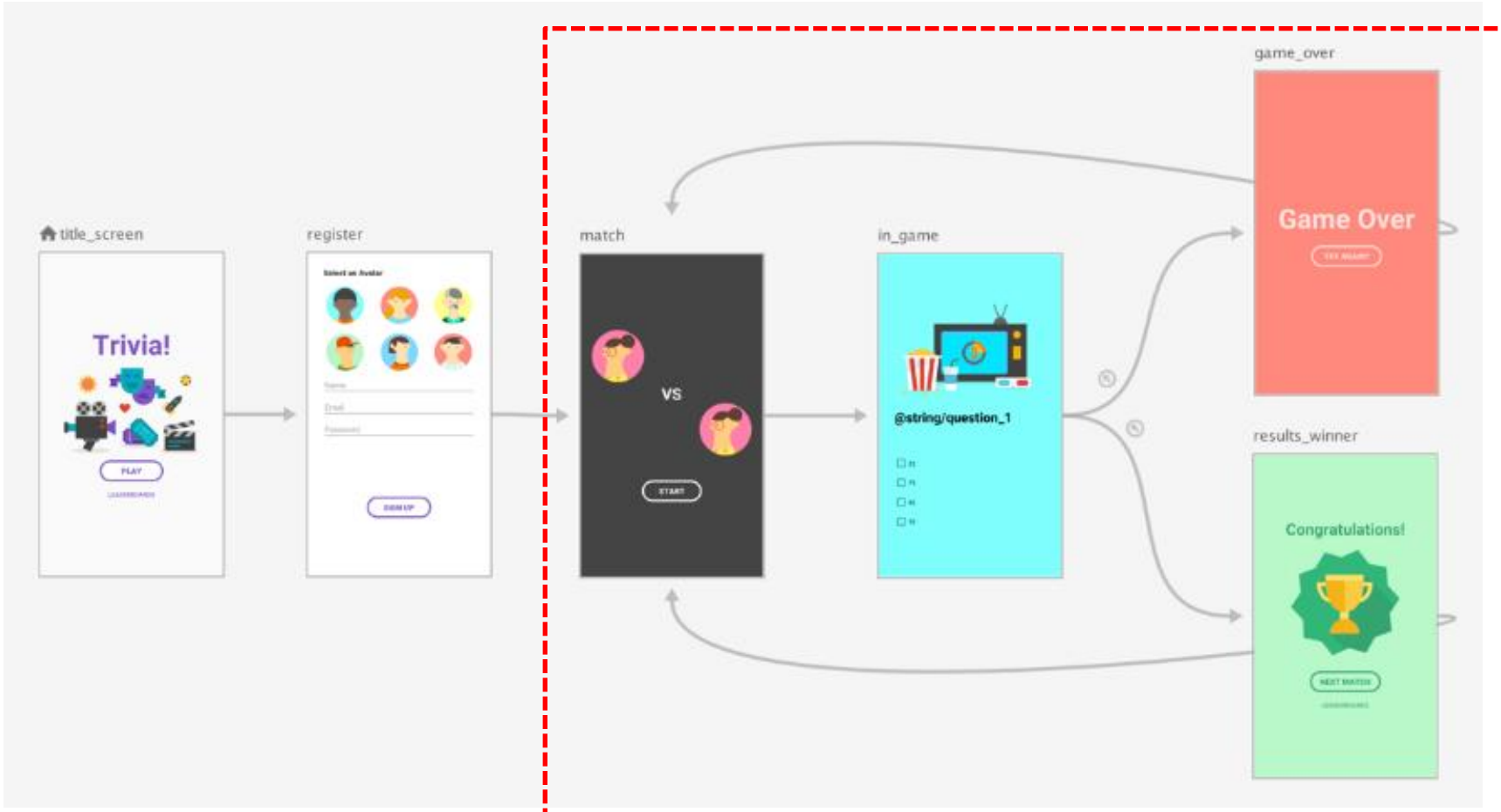
- BottonNavigation 과 BottomNavigationItem 이용
  - BottomNavigation은 서로 독립적인 화면 구성시 주로 사용

```
val items = listOf(
    Screen.Profile,
    Screen.FriendsList,
)
```

```
val navController = rememberNavController()
Scaffold(
  bottomBar = {
    BottomNavigation {
      val navBackStackEntry by navController.currentBackStackEntryAsState()
      val currentDestination = navBackStackEntry?.destination
      items.forEach { screen ->
        BottomNavigationItem(
          icon = { Icon(Icons.Filled.Favorite, contentDescription = null) },
          label = { Text(stringResource(screen.resourceId)) },
          selected = currentDestination?.hierarchy?.any { it.route == screen.route } == true,
          onClick = {
            navController.navigate(screen.route) {
              // Pop up to the start destination of the graph to
              // avoid building up a large stack of destinations
              // on the back stack as users select items
              popUpTo(navController.graph.findStartDestination().id) {
                saveState = true
              }
              // Avoid multiple copies of the same destination when
              // reselecting the same item
              launchSingleTop = true
              // Restore state when reselecting a previously selected item
              restoreState = true
            }
          }
        )
      }
    }
  }
) { innerPadding ->
  NavHost(navController, startDestination = Screen.Profile.route, Modifier.padding(innerPadding)) {
    composable(Screen.Profile.route) { Profile(navController) }
    composable(Screen.FriendsList.route) { FriendsList(navController) }
  }
}
```

Nested Navigation

- 중첩 그래프
  - 앱 내의 여러 개의 흐름을 표현할 때 사용



Nested Navigation

- navigation 함수를 이용해서 새로운 흐름 생성

```
NavHost(navController, startDestination = "title_screen") {
    composable("title_screen") {
        TitleScreen(
            onPlayClicked = { navController.navigate("register") },
            onLeaderboardsClicked = { /* Navigate to leaderboards */ }
        )
    }
    composable("register") {
        RegisterScreen(
            onSignUpComplete = { navController.navigate("gameInProgress") }
        )
    }
    navigation(startDestination = "match", route = "gameInProgress") {
        composable("match") {
            MatchScreen(
                onStartGame = { navController.navigate("in_game") }
            )
        }
        composable("in_game") {
            InGameScreen(
                onGameWin = { navController.navigate("results_winner") },
                onGameLose = { navController.navigate("game_over") }
            )
        }
        composable("results_winner") {
            ResultsWinnerScreen(
                onNextMatchClicked = {
                    navController.navigate("match") {
                        popUpTo("match") { inclusive = true }
                    }
                },
                onLeaderboardsClicked = { /* Navigate to leaderboards */ }
            )
        }
```

NavGraphBuilder 확장함수

```
fun NavGraphBuilder.addNestedGraph(navController: NavController) {
    navigation(startDestination = "match", route = "gameInProgress") {
        composable("match") {
            MatchScreen(
                onStartGame = { navController.navigate("in_game") }
            )
        }
        composable("in_game") {
            InGameScreen(
                onGameWin = { navController.navigate("results_winner") },
                onGameLose = { navControl
            )
        }
        composable("results_winner") {
            ResultsWinnerScreen(
                onNextMatchClicked = { na
                onLeaderboardsClicked = {
            )
        }
        composable("game_over") {
            GameOverScreen(
                onTryAgainClicked = { nav
            )
        }
    }
}
```

```
@Composable
fun MyApp() {
    val navController = rememberNavController()
    NavHost(navController, startDestination = "title_screen") {
        composable("title_screen") {
            TitleScreen(
                onPlayClicked = { navController.navigate("register") },
                onLeaderboardsClicked = { /* Navigate to leaderboards */ }
            )
        }
        composable("register") {
            RegisterScreen(
                onSignUpComplete = { navController.navigate("gameInProgress") }
            )
        }

        // Add the nested graph using the extension function
        addNestedGraph(navController)
    }
}
```

Side Effect

- Side Effect?
  - Composable에서 자신이 아닌 외부의 State에 영향을 주는 것
    - Composable 범위 밖에서 앱 상태를 변화 시킴
    - 예) 양방향 의존성으로 인해 예측할 수 없는 Effect가 생길 수 있음
- LaunchedEffect
  - Composable 함수에서 suspend 함수를 실행하기 위해 사용
- DisposableEffect
  - Composable이 Dispose될 때 정리되어야 할 Side Effect를 처리하기 위해 사용
- SideEffect
  - Composable의 state를 Compose에서 관리하지 않는 객체와 공유하기 위해 사용

Side Effect

- rememberCoroutineScope
  - Composable의 CoroutineScope를 참조하여 외부에서 실행할 수 있도록 함
- rememberUpdateState
  - Launched Effect는 컴포저블의 State가 변경되면 재실행되는데 재실행되지 않아도 되는 State를 정의하기 위해 사용
- produceState
  - Composable의 State가 아닌 것을 Composable의 state로 변환
- derivedStateOf
  - State를 다른 state로 변화하기 위해 사용하며, Composable은 변환된 State에만 영향을 받음
- snapshotFlow
  - Composable의 State를 Flow로 변환

LaunchedEffect

- 컴포저블에서 suspend 함수를 실행해주는 컴포저블
  - Key가 바뀔 때만 LaunchedEffect의 suspend 함수를 실행

```
fun LaunchedEffect(
    key1: Any?,
    block: suspend CoroutineScope.() -> Unit
) {
    val applyContext : CoroutineContext = currentComposer.applyCoroutineContext
    remember(key1) { LaunchedEffectImpl(applyContext, block) }
}
```

DisposableEffect

- 컴포저블이 Dispose된 후에 정리해야 할 Side Effect가 있는 경우에 사용되는 Effect

```
fun DisposableEffect(
    key1: Any?,
    effect: DisposableEffectScope.() -> DisposableEffectResult
) {
    remember(key1) { DisposableEffectImpl(effect) }
}
```

  - 처음에는 초기화 로직만 수행한 후, 키 값이 바뀔 때마다 onDispose 블록을 호출한 후 초기화 로직 수행

```
DisposableEffect(key1 = key){
    // 초기화 로직 작성
    onDispose {
    // Dispose 되어야 하는 Effect 제거
    }
}
```

수고하셨습니다.