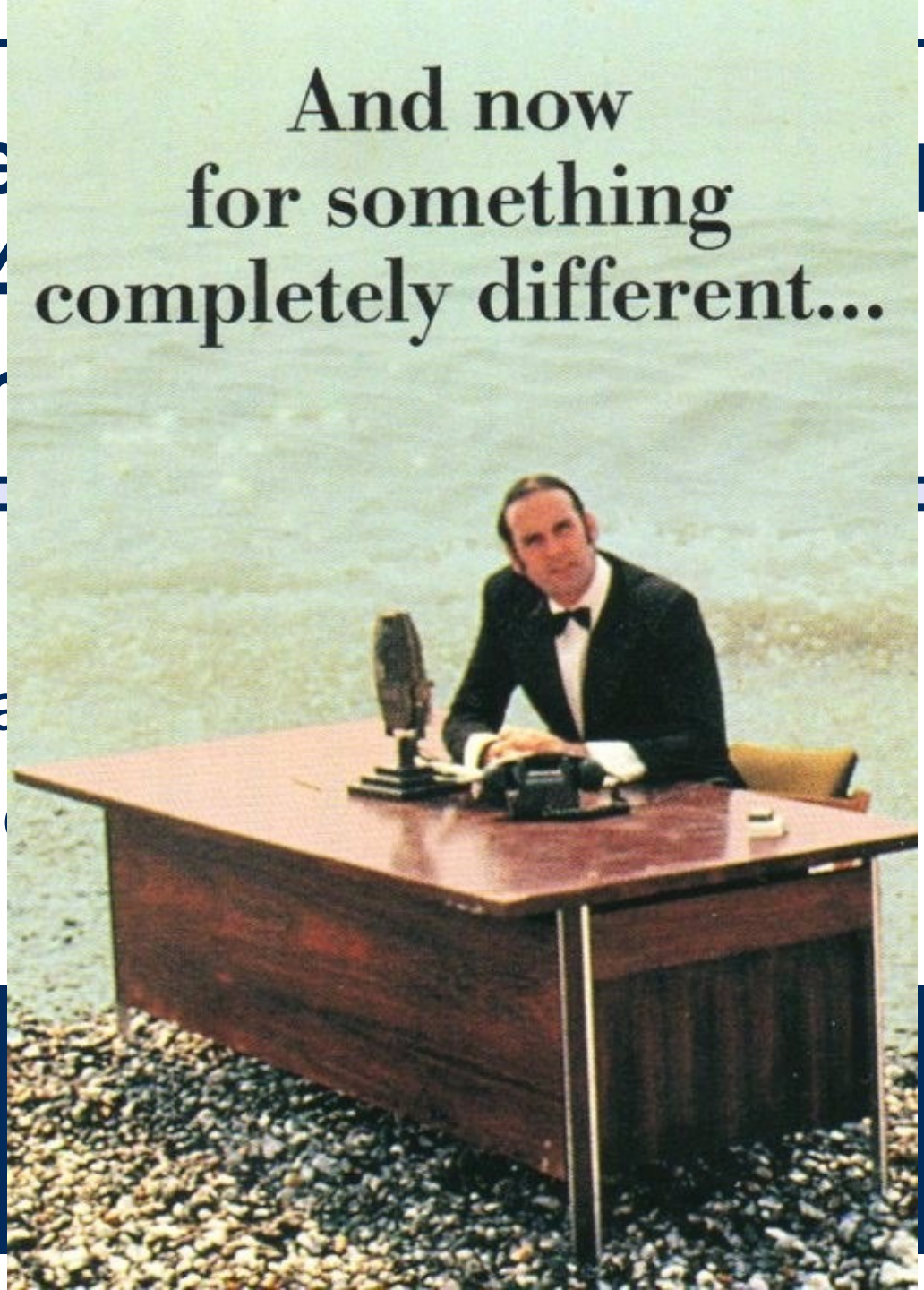
Mobile Application Development
CSCI 448
Lecture 15

NavGraphs

Arguments

Mobile
CSCI 4
Lectur



NavGra

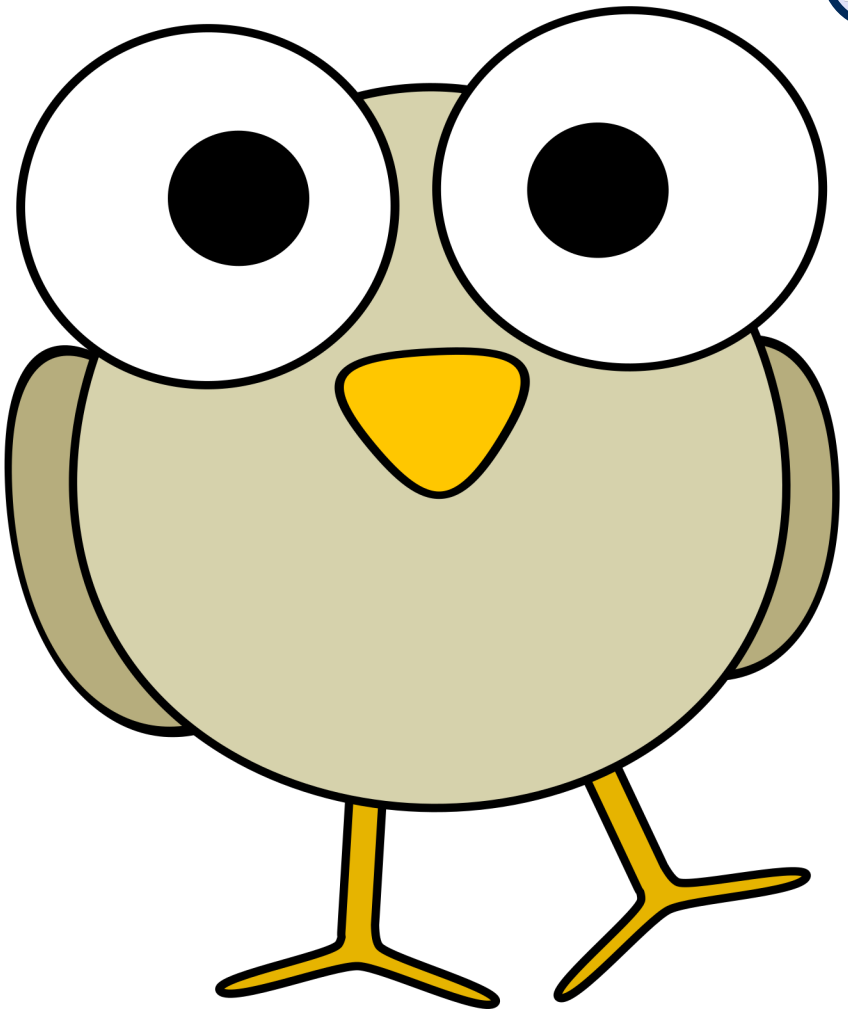
Argum

pment

Previously in CSCI 448

- Navigation
  - `NavHost` composable
  - Create NavGraph via *`composable`* destination builder method
  - NavController performs navigation to destinations

Questions?



Learning Outcomes For Today

- Discuss how to implement Jetpack Navigation in the Compose framework
- Describe how NavGraphs implement the Builder Design Pattern
- Discuss the benefits of using a sealed class and implement an abstract Navigation Destination

On Tap For Today

- Navigation Components
- Object-Oriented Design

On Tap For Today

- Navigation Components

- Object-Oriented Design

Navigation Components

- NavController
- NavHost
- NavGraph

Navigation Components

```
NavHost(navController = navController, startDestination = "myNavGraph") {
  navigation (route = "myNavGraph", startDestination = "myRoute") {
    composable(route = "myRoute") {
        Text("This is my destination!")
        Button( onClick = { navController.navigate("listScreen") } ) {
            Text("Go To List")
        }
    }
    composable(route = "listScreen") {
        ListScreen(list) { navController.navigate("detailScreen") }
    }
    composable(route = "detailScreen") {
        DetailScreen(detailObject = ???)
    }
  }
}
```

Arguments Problem

- Screen B may need some parameter to be created

```
@Composable
fun DetailScreen(detailObject: DetailObject) {
  Column {
    Text( text = detailObject.name )
    Text( text = detailObject.address )
  }
}
```

NavGraph Arguments: Declaring

1. Specify within route
2. Create argument name to type association

```
composable(
  route = "detail/{id}",
  arguments = listOf( navArgument("id") { type = NavType.StringType } )
) {
    Text("This is my destination!")
}
```

NavGraph Arguments: Specifying

- When navigating, provide value to apply

```
navController.navigate("detail/123")
```

NavGraph Arguments: Retrieving

- BackStackEntry contains list of argument values

```
composable(
  route = "detail/{id}",
  arguments = listOf( navArgument("id") { type = NavType.String } )
) { backStackEntry ->

    val argVal = backStackEntry.arguments?.getString("id", "") ?: ""
    Text("This is my destination with $argVal!")
}
```

NavGraph Arguments: Retrieving

- BackStackEntry contains list of argument values

```
composable(
  route = "detail/{id}",
  arguments = listOf( navArgument("id") { type = NavType.String } )
) { backStackEntry ->

    val argVal = backStackEntry.arguments?.getString("id", "") ?: ""
    val specificObject = /* serviceToDoLookupById(id = argVal) */

    if(specificObject != null)
      DetailScreen(detailObject = specificObject)
}
```

Nested NavGraphs

```
NavGraph(navController = navController, startDestination = "homepage") {
    navigation(startDestination = "myRoute", route = "homepage") {
        composable(route = "myRoute") {
            Text("Welcome!")
            Button( onClick = { navController.navigate("info") } ) {
                Text("Start")
            }
        }
    }
    navigation(startDestination = "listScreen", route = "info") {
        composable(route = "listScreen") {
            ListScreen(list) { id -> navController.navigate("detail/$id")
        }
        composable(
          route = "detail/{id}",
          arguments = listOf( navArgument("id") { type = NavType.String } )
        ) { backStackEntry ->
            val argVal = backStackEntry.arguments?.getString("id", "") ?: ""
            Text("This is my destination with $argVal!")
        }
    }
}
```

On Tap For Today

- Navigation Components
- Object-Oriented Design

Concrete NavGraphs

```
NavGraph(navController = navController, startDestination = "homepage") {

    navigation(startDestination = "myRoute", route = "homepage") {
        composable(route = "myRoute") {
            Text( text = "This is my home destination!"),
             modifier = Modifier.clickable { navController.navigate("info") }
            )
        }
    }

    navigation(startDestination = "listScreen", route = "info") {
        composable(route = "listScreen") {
            ListScreen(list) { id -> navController.navigate("detail/$id")
        }
        composable(
          route = "detail/{id}",
          arguments = listOf( navArgument("id") { type = NavType.String } )
        ) { backStackEntry ->
            val argVal = backStackEntry.arguments?.getString("id", "") ?: ""
            Text("This is my destination with $argVal!")
        }

    }
}
```

Design Principles

1. Write Once Read Many
2. Separation of concerns
3. Composition over inheritance

Design Principles

1. Write Once Read Many
2. Separation of concerns
3. Composition over inheritance
4. Program to an interface, not an implementation
5. Encapsulate what varies

Abstract NavGraphs

```
NavHost(navController = navController, startDestination = "...") {
    // foreach navigation
    //    set composable startDestination and route to navgraph
    //    foreach composable
    //      set route and arguments
    //      specify content
}
```

Abstract NavGraphs

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

    allNavGraphs.forEach { navGraph ->

        navigation(
          startDestination = navGraph.start,
          route = navGraph.root
        ) {

            navGraph.allDestinations.forEach { destination ->

                composable(
                  route = destination.route,
                  arguments = destination.args
                ) { backStackEntry ->

                    destination.Content(navController, backStackEntry)

                }

            }

        }

    }
}
```

Abstract Destinations

- Same process for abstract NavGraphs
- Create a sealed interface to represent a Destination
  - Our Destinations are Screens
  - Need: route, arguments, Content
  - Constitutes a specification
- Will call it **`IScreenSpec`**

Sealed Interface

```
sealed interface IScreenSpec {

  val route: String
  val arguments: List<NamedNavArgument>
  fun navigateTo(vararg args: String?): String

  @Composable fun Content(navController: NavController,
                          navBackStackEntry: NavBackStackEntry)

}
```

Create Concrete Instance

```
sealed interface IScreenSpec {

  val route: String
  val arguments: List<NamedNavArgument>
  fun navigateTo(vararg args: String?): String

  @Composable fun Content(navController: NavController,
                          navBackStackEntry: NavBackStackEntry)

}

object DetailScreenSpec : IScreenSpec {

  private const val ARG = "id"
  override val route = "detail/{$ARG}"
  override val arguments: List<NamedNavArgument> = listOf(
      navArgument(ARG) { type = NavType.String }
  )
  fun navigateTo(vararg args: String?): String = "detail/${args[0]}"

  @Composable
  fun Content(navController: NavController,
                          navBackStackEntry: NavBackStackEntry) {
      DetailScreen(...)
  }

}

// and make ListScreenSpec
```

Store All Concrete Instances

```
sealed interface IScreenSpec {

  companion object {
    val allScreens = listOf(
        DetailScreenSpec,
        ListScreenSpec
    )
    const val root = "list"
    val startDestination = ListScreenSpec.route
  }

  val route: String
  val arguments: List<NamedNavArgument>
  fun navigateTo(vararg args: String?): String

  @Composable fun Content(navController: NavController,
                          navBackStackEntry: NavBackStackEntry)

}
```

Abstract NavGraphs

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

    IScreenSpec.allScreens.forEach { screen ->

        composable(
          route = screen.route,
          arguments = screen.arguments
        ) { backStackEntry ->

            screen.Content(navController, backStackEntry)

        }

    }
}
```

Maintenance Problem?

```
sealed interface IScreenSpec {

  companion object {
    val allScreens = listOf(
        DetailScreenSpec,
        ListScreenSpec
    )
    const val root = "list"
    val startDestination = ListScreenSpec.route
  }

  val route: String
  val arguments: List<NamedNavArgument>
  fun navigateTo(vararg args: String?): String

  @Composable fun Content(navController: NavController,
                          navBackStackEntry: NavBackStackEntry)

}
```

Sealed Class / Interface

- All direct subclasses are known at compile time, no other subclasses may appear after module is compiled
  - Third-party clients cannot extend a sealed class in their code
- Types are from a known limited set
- All direct subclasses must be declared in the same package

Maintenance Solved via Reflection

```
sealed interface IScreenSpec {

  companion object {
    val allScreens = IScreenSpec::class.sealedSubclasses.map { it.objectInstance }
    const val root = "home"
    val startDestination = HomeScreenSpec.route
  }

  val route: String
  val arguments: List<NamedNavArgument>
  fun navigateTo(vararg args: String?): String

  @Composable fun Content(navController: NavController,
                          navBackStackEntry: NavBackStackEntry)

}
```

Abstract NavGraphs

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

  navigation(
    route = IScreenSpec.root,
    startDestination = IScreenSpec.startDestination
  ) {

      IScreenSpec.allScreens.forEach { screen ->

        composable(
          route = screen.route,
          arguments = screen.arguments
        ) { backStackEntry ->

            screen.Content(navController, backStackEntry)

        }

      }

  }
}
```

Design Pattern #7: Template Method

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Participants:
  - `AbstractClass`: defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm AND implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects
  - `ConcreteClass`: implements the primitive operations to carry out subclass-specific steps of the algorithm

Android Design Patterns

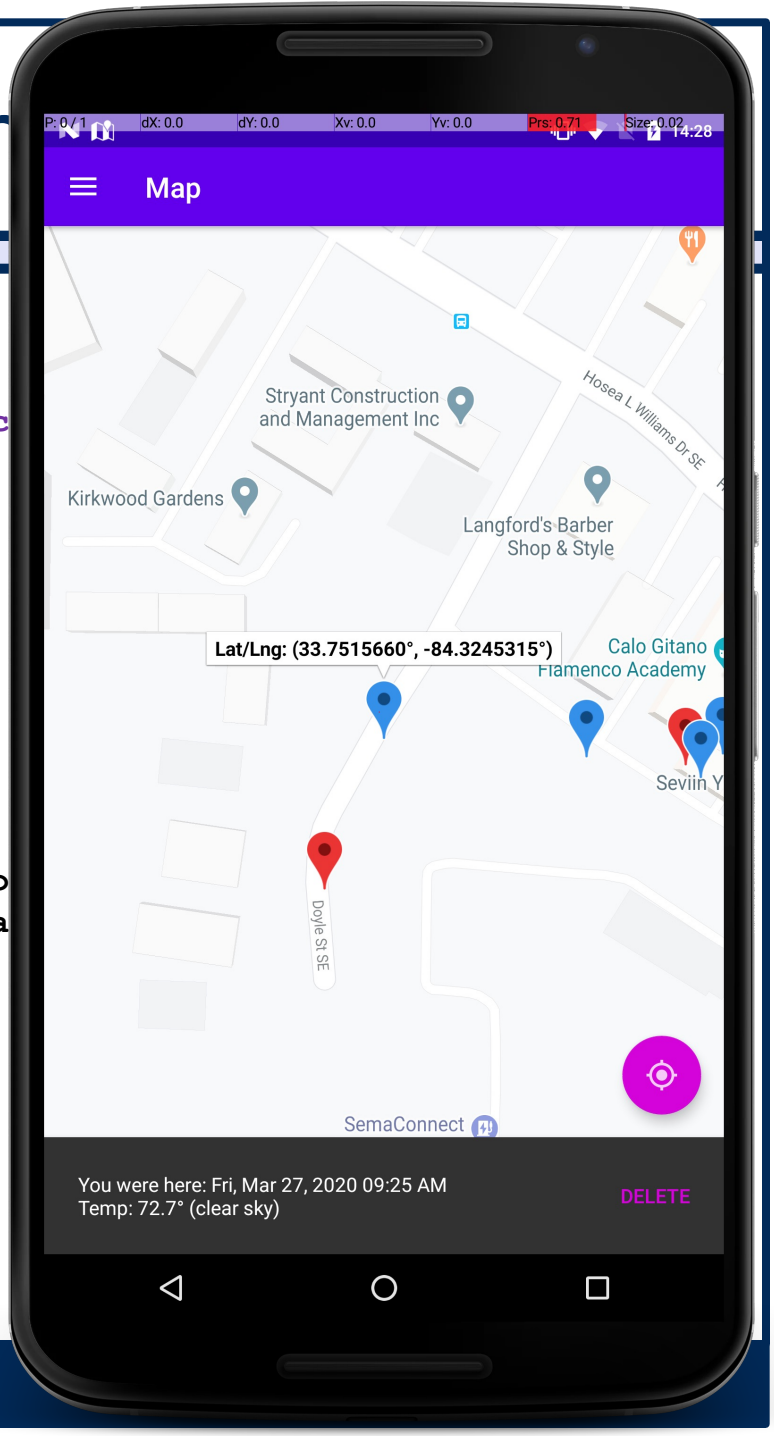
- Behavioral Patterns
  1. Command – UI Event Handling
  2. Observer – State
  3. Template Method - IScreenSpec
- Creational Patterns
  4. Builder – Compose NavGraph
  5. Factory – ViewModelFactory
  6. Singleton – ViewModelProvider, Repository
- Structural Patterns
  7. Decorator – View Model

Screen Specification Will Expand

```
sealed interface IScreenSpec {

  companion object {
    val allScreens = IScreenSpec::class.sealedSubclasses.map { it.objectInstance }
    const val root = "home"
    val startDestination = HomeScreenSpec.route
  }

  val route: String
  val arguments: List<NamedNavArgument>
  fun navigateTo(vararg args: String?): String

  @get:StringRes val titleId: Int

  @Composable fun TopAppBarActions()

  @Composable fun Content(navController: NavController,
                          navBackStackEntry: NavBackStackEntry)

  @Composable fun FABContent()

}
```

Screen Specification

```
sealed interface IScreenSpec {

  companion object {
    val allScreens = IScreenSpec::class.sealedSubc
    const val root = "home"
    val startDestination = HomeScreenSpec.route
  }

  val route: String
  val arguments: List<NamedNavArgument>
  fun navigateTo(vararg args: String?): String

  @get:StringRes val titleId: Int

  @Composable fun TopAppBarActions()

  @Composable fun Content(navController: NavContro
                          navBackStackEntry: NavBa

  @Composable fun FABContent()

}
```



On Tap For Today

- Navigation Components
- Object-Oriented Design

To Do For Next Time

- Continue on Lab4 - due Thu Feb 23

- A2 posted
- Lab05 coming

- Alpha Release due Mon Mar 13
  - Have screens and navigation in place