CMPUT 301 Lab 2

OOP Principles

Lab 2 Overview

- Review Kotlin OOP ideas used in Android
- Understand the basics of an Android screen
- Learn how Compose displays UI using composable functions
- Lab Demo

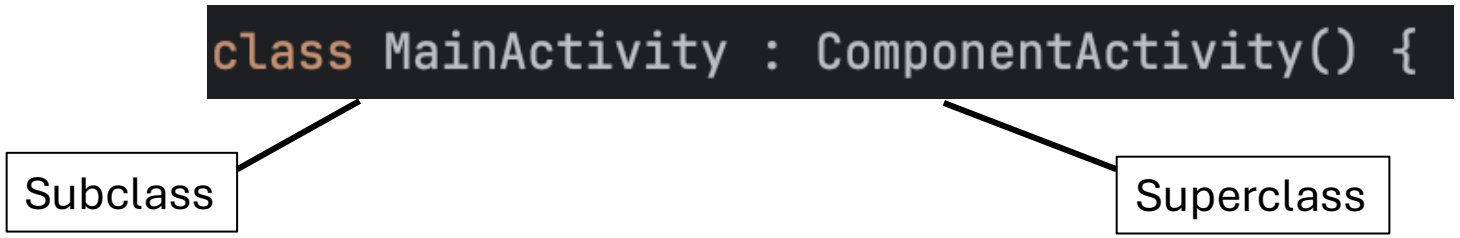
OOP Review

- Android screens are usually represented by an activity, such as MainActivity
- Android provides parent classes, such as ComponentActivity
- You inherit Android behaviour and override parts of it
- In Compose, UI is displayed using setContent { ... }
- You store and protect app data using properties and functions

```
class MainActivity : ComponentActivity() {  1 Usage
    // MainActivity inherits Android activity behaviour from ComponentActivity
```

Inheritance

- Inheritance means one class gets behaviour from another class
- Superclass (parent)
  - ComponentActivity is the Android class being inherited from
- Subclass (child)
  - MainActivity is our app's screen class
- The subclass will inherit all fields and methods
  (basic constructor, attributes, and all user-defined functions)



Inheritance

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enableEdgeToEdge()
    setContent {
```

- MainActivity inherits from ComponentActivity
  - onCreate runs when the screen is created
  - super.onCreate(...) calls the parent class setup
  - setContent { ... } tells Compose what UI to display
- Then we add our app-specific setup

Polymorphism

- **Polymorphism** means you can call the same action on different objects, and each one does it in its own way
- Animal is the **superclass** (parent class)
- Dog is the **subclass** (child class)
- Dog overrides speak() to provide its own behaviour
- The variable type is Animal, but the actual object is Dog
- Kotlin calls Dog's version of speak() at runtime

```kotlin
open class Animal {  2 Usages
    1 Override
    open fun speak() {  1 Usage
        println("some noise")
    }
}

open class Dog : Animal() {  1 Usage
    override fun speak() {
        println("woof")
    }
}

fun main() {
    val animal: Animal = Dog()
    animal.speak() // woof
}
```

Encapsulation

- When do I use private (or protected) variables?
  - Whenever you want to hide data from other classes
  - Whenever outside code should not access or change that data directly
- Keep mutable data private
  - Outside code should not modify it directly
- Use functions to update app state instead of modifying properties from anywhere in the app
- Encapsulation helps keep the displayed UI consistent with the app data

```
class CityRepository {  1 Usage
    private val _cities = mutableListOf(  2 Usages
        "Edmonton", "Vancouver", "Calgary"
    )

    val cities: List<String>  1 Usage
        get() = _cities

    fun addCity(city: String) {
        _cities.add(city)
    }
}
```

List Example and Exercise (demo)