IN2000
Software Engineering med prosjektarbeid

Tilstand, Coroutines og intro til app-arkitektur

Menti: **3397 5892**

Sondre Bader Wang og Steffen Almås

Spørsmål - Menti

- Spørsmål underveis?
- Menti: **3397 5892**

Oppsummering av forrige uke

Menti: **3397 5892**

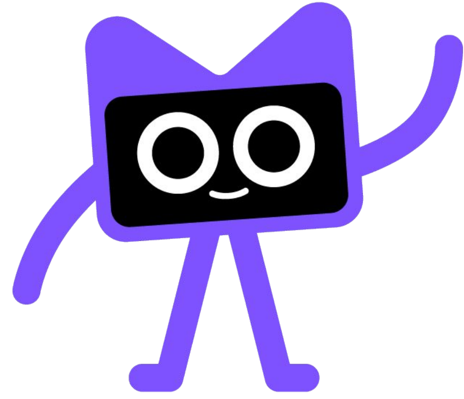
Oppsummering - Generelt

- De første ukene i IN2000 går fort!
- Du behøver ikke være ekspert på teknologiene
- Grunnlaget til prosjektarbeidet
- Det tekniske er en modningsprosess

- Obligatorisk innlevering 1, *frist tirsdag 30. januar*
- Teaminnmelding, *frist fredag 2. februar*

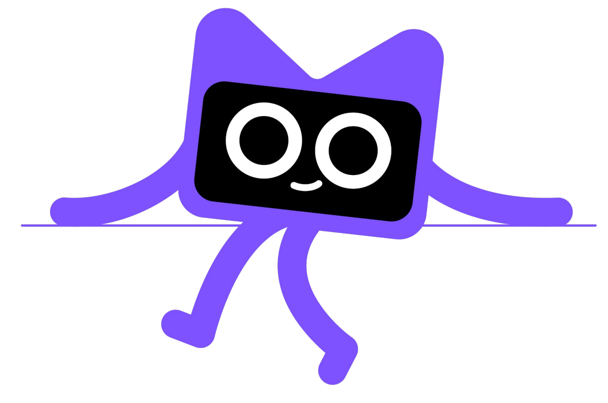
Oppsummering - Kotlin

- Programmeringsspråk laget av JetBrains
- Offisielt språk for Android-utvikling
- Laget for å “fikse” det som er dårlig i Java



Hva har vi gått gjennom til nå? - Kotlin

- Variabler
- Mutable / Immutable
- String templates
- Interoperabilitet med Java
- Funksjoner
- Kontrollflyt
- Null safety
- Kort om klasser og objekter



Hva har vi gått igjennom til nå - Android

- Hvordan kjører vi android-apper
- Hva er Jetpack Compose
- Composable-funksjoner og noen eksempler på dette
- Previews
- Compose-state og remember
- Unidirectional Data Flow
- Gradle
- Navigasjon

Logcat

Menti: **3397 5892**

Logcat - Generelt

- Logge meldinger til terminal under kjøring
  - erstatter `println(“jeg er her i appen nå”)`
- Ryddig og strukturert
- **`Log.v()`** har 2 parametre
  - Tag : String
  - Message : String

```
private const val TAG = "MainActivity"
Log.i(TAG, "Item number $position")
```

Logcat - Ulike nivåer logging

- `Log.e(String, String)` → Error
- `Log.w(String, String)` → Warning
- `Log.i(String, String)` → Information
- `Log.d(String, String)` → Debug
- `Log.v(String, String)` → Verbose

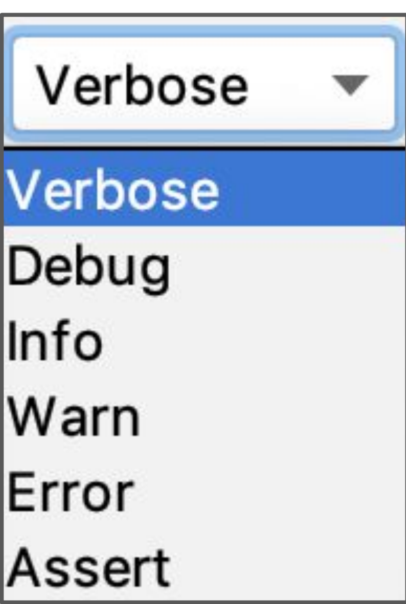
Logcat - Ulike nivåer logging

- `Log.e(String, String)` → Error
- `Log.w(String, String)` → Warning
- `Log.i(String, String)` → Information
- `Log.d(String, String)` → Debug
- `Log.v(String, String)` → Verbose

- `Log.wtf(String, String)` → What a Terrible Failure

Logcat - Filtrere på nivå

- Filtrere på nivå i Logcat i Android Studio
- Viser kun den typen som er valgt
  - Standard `println()` er en del av Verbose



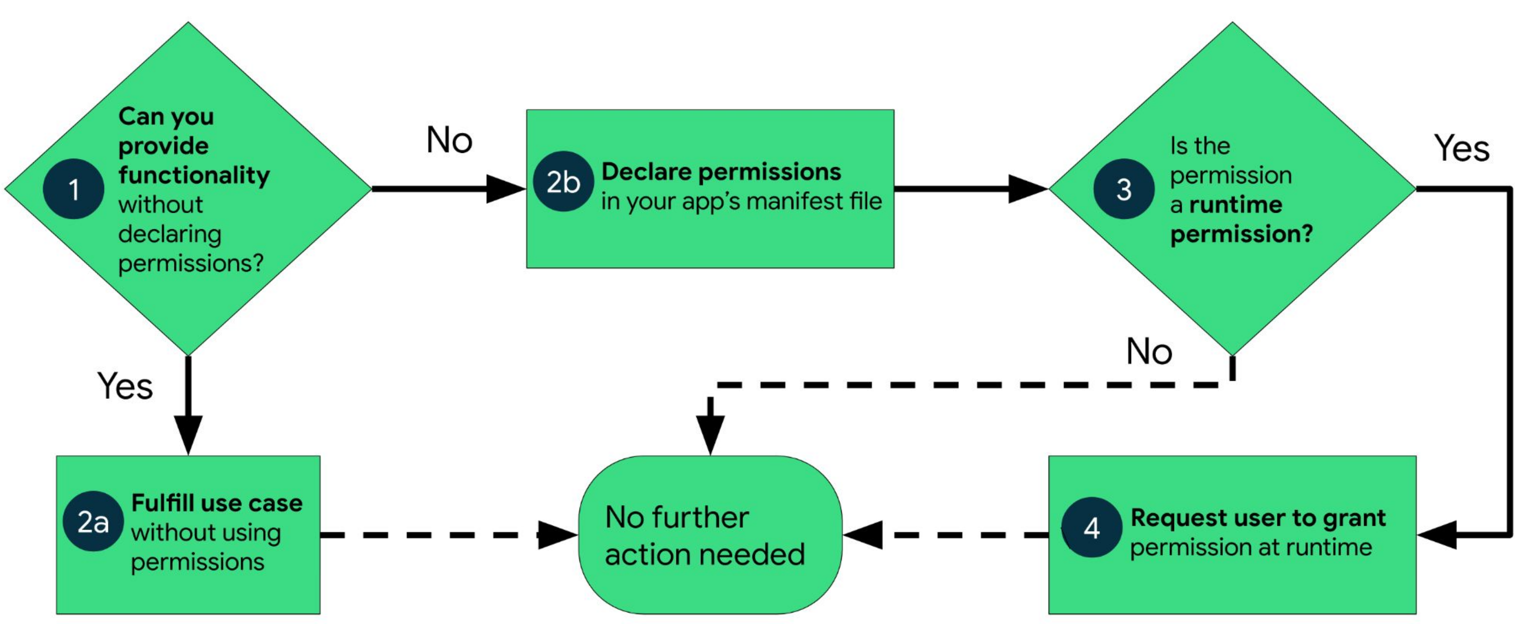
Permissions

Menti: **3397 5892**

Permissions

- Install-time permissions
  - Deklareres i manifestet
  - Hente data fra internett
  - Full nettverkstilgang
  - Unngå at telefonen “sover”
- Runtime permissions (dangerous permissions)
  - Du kan ikke anta at disse tilgangene har blitt gitt tidligere, sjekk de, og hvis det trengs, forespør de før hver gang du skal benytte deg av de
  - Disse gir ofte tilgang til privat brukerdata med potensielt sensitiv informasjon
  - Kameratilgang
  - Tilgang til posisjon
- Mange ting som å ta bilder, pause mediaavspilling osv kan gjøres uten å deklarere noen permissions

Permissions-workflow



https://developer.android.com/guide/topics/permissions/overview

Kotlin: Mer om klasser og objekter

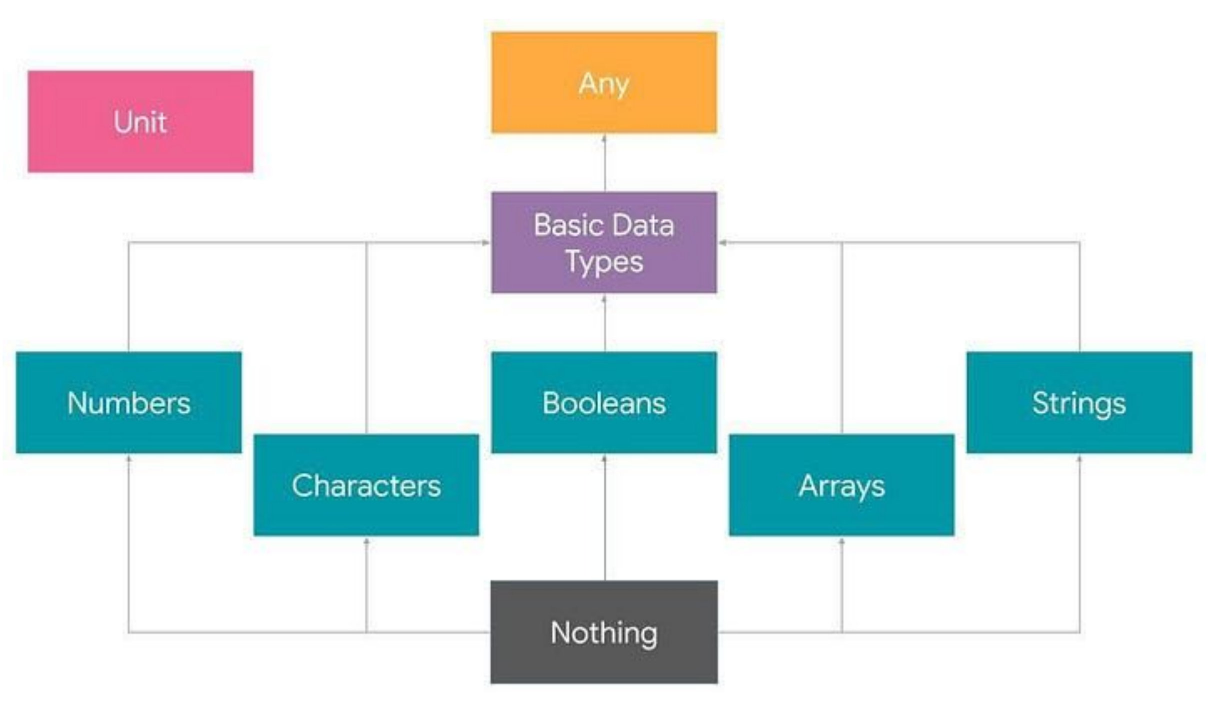
Menti: **3397 5892**

Klasser og objekter - Menti spm. fra i fjor

- "Er variabler typer eller objekter?"

- **Variabler er objekter**
  - Og skal se mer på det nå!

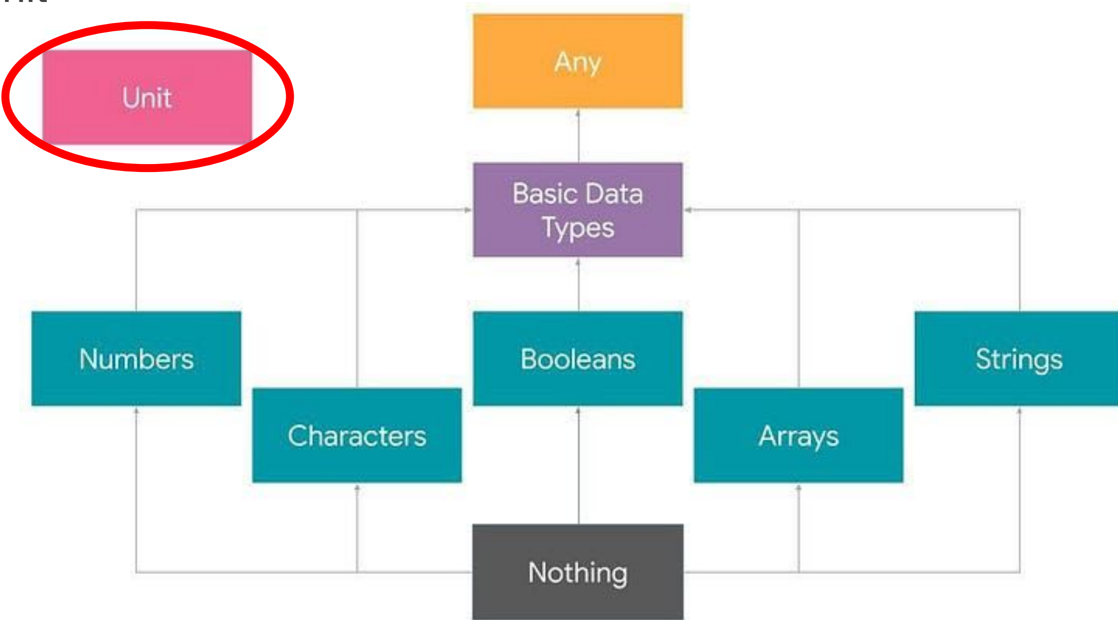
Klasser og objekter - Kotlin klassehierarki



Klasser og objekter - Unit

- `Unit` = det “samme” som `void` i Java
  - Forskjell: I Java er ikke dette representert ved en klasse, i Kotlin er det
  - Trenger ikke eksplisitt si “Unit” når metode ikke returnerer noe

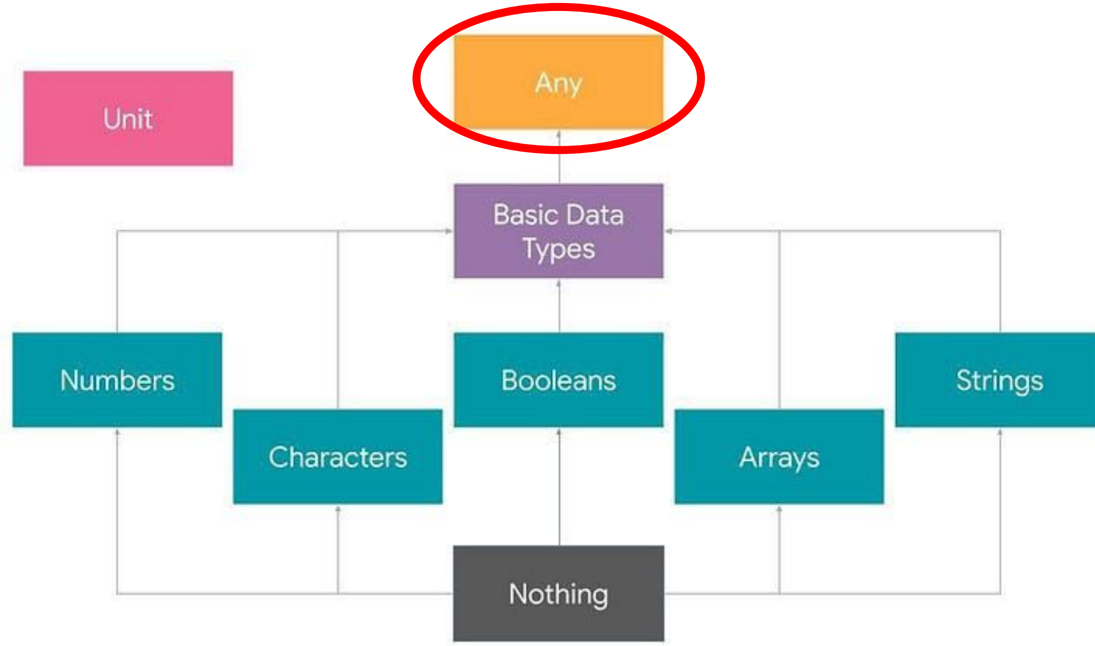
```kotlin
fun main() {
    printHello()
}

fun printHello(): Unit {
    println("Hello World")
}
```



Klasser og objekter - Any

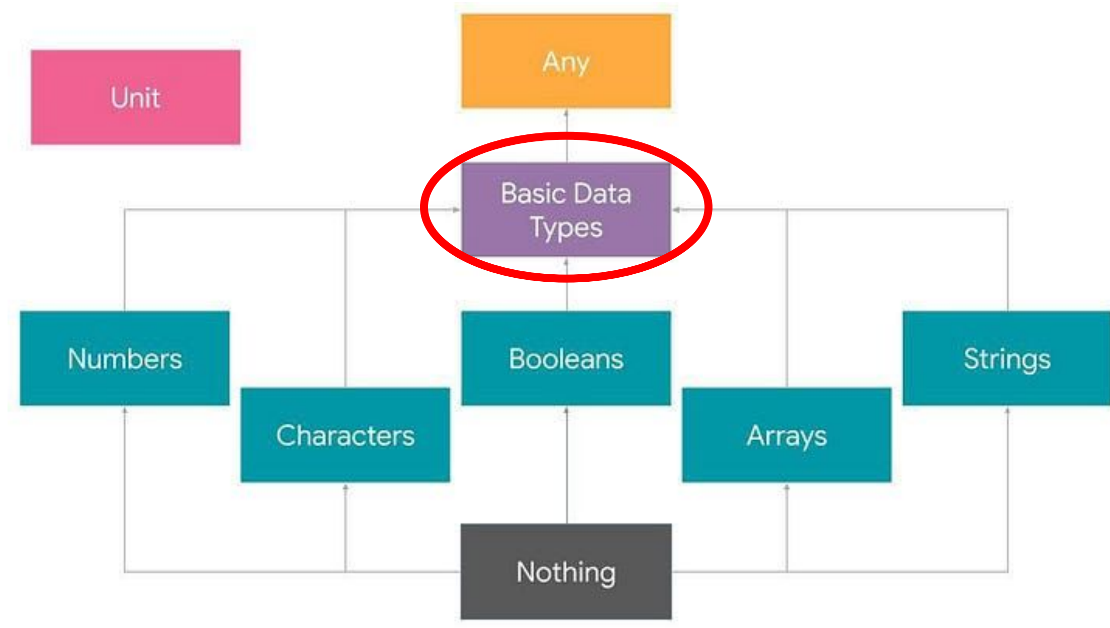
- Any = superklassen til alle klasser i Kotlin
  - Samme som Object i Java

```
fun main() {
    val text = getHello()
    val number = getNumber()
    println("$text $number you")
    >> Hello 2 you
}

fun getHello(): Any = "Hello"
fun getNumber(): Any = 2
```



Klasser og objekter - Basic types

- Basic types som dere kjenner fra før :)
  - Numbers
  - Characters
  - Booleans
  - Arrays
  - Strings



Klasser og objekter - Nothing

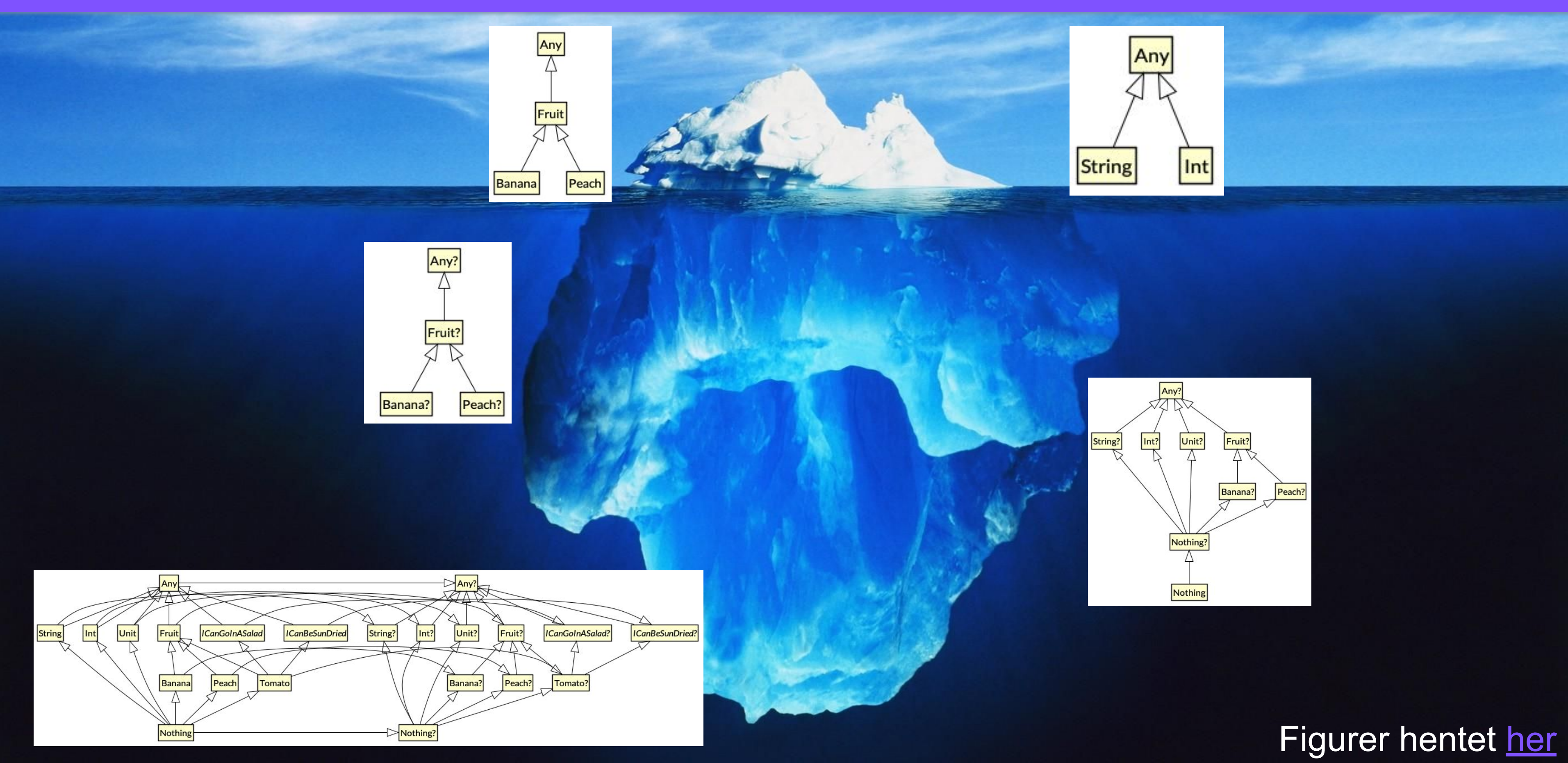
- `Nothing` = brukes for å representere en verdi som aldri kommer til å eksistere. F.eks. returtype i Exceptions eller ved terminering av main

Klasser og objekter - Nothing (forts.)

- `Nothing` = brukes for å representere en verdi som aldri kommer til å eksistere. F.eks. returtype i Exceptions eller ved terminering av main.

```
fun main() {
    println(divide(2, 0))
}

fun error(message: String): Nothing {
    throw IllegalArgumentException(message)
}

fun divide(a: Int, b: Int): Int = if (b == 0) error("Division by zero") else a / b
```

Klasser og objekter - Kotlin klassehierarki, Iceberg



Figurer hentet [her]

Klasser og objekter - enum

- `enums` = brukes for å representere et sett med kjente verdier, f.eks.:
  - Ukedager
  - Kompassretninger
  - Enheter 🤠
- Øker "type safety"

Klasser og objekter - enum (fors.)

```
enum class Day {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}

fun main() {
    val today = Day.MONDAY
    println(if (today == Day.SATURDAY || today == Day.SUNDAY) "Weekend!" else "Weekday...")
}
```

Tilstand og produksjon av tilstand

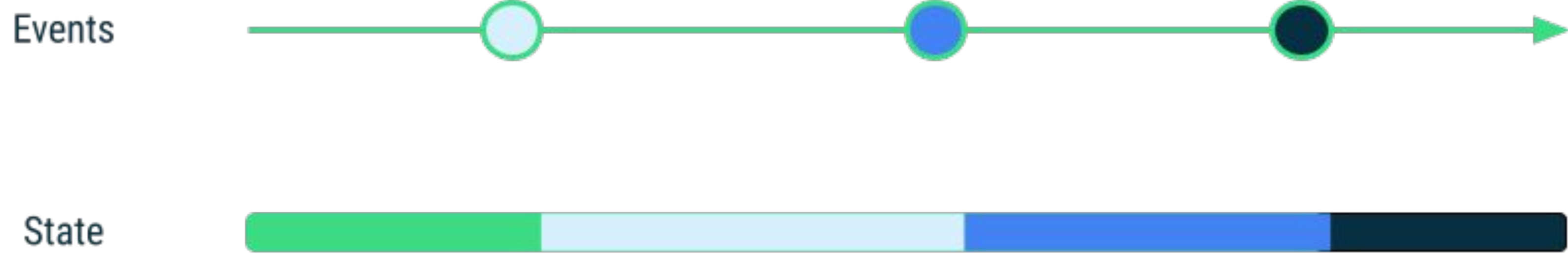
Menti: **3397 5892**

Tilstand - repetisjon fra sist

- En app sin tilstand er enhver verdi som kan endre seg over tid
- For å håndtere tilstand i Compose benyttes den observerbare typen State
- For å lagre ting i komposisjonen benytter vi oss av `remember`
- Vi kan flytte tilstand ut av en composable med bruk av state hoisting
- Vi kan skille visning av brukergrensesnitt og produksjon av tilstand fra hverandre

Produksjon av tilstand

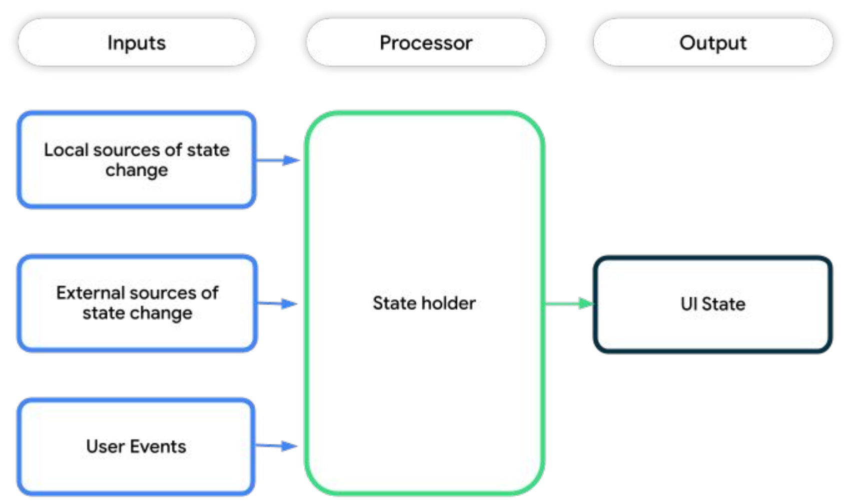
- Brukergrensesnittet er en visuell representasjon av applikasjonens tilstand
- Events forårsaker tilstandsendringer



Produksjon av tilstand (forts.)

Events kan komme fra:

- **Brukere** ved at de interagerer med appens brukergrensesnitt
- **Ekstern input**, f.eks. at en datakilde er ferdig med å laste ned data



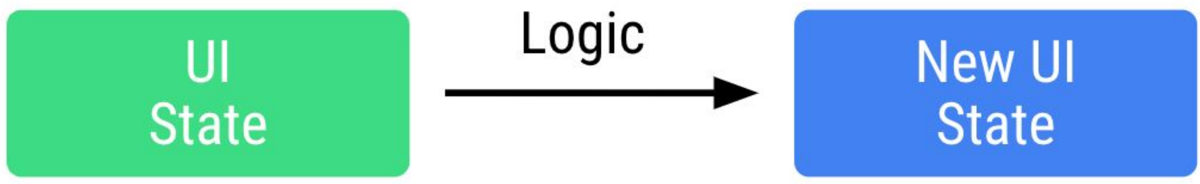
UI state

- UI state er den egenskapen som beskriver brukergrensesnittet.
- **Screen UI State** som er *hva* du trenger å vise på skjermen.
  - For eksempel kan en NewsUiState klasse inneholde nyhetsartikler og annen informasjon nødvendig for å vise frem brukergrensesnittet.
- **UI element State** som refererer til egenskaper som er iboende UI-elementer og har en innflytelse på hvordan de er vist.
  - F.eks. at et UI-element kan bli vist eller skjult.
  - Eksempel på dette er ScaffoldState for Scaffold composablen.

Logikk i apper

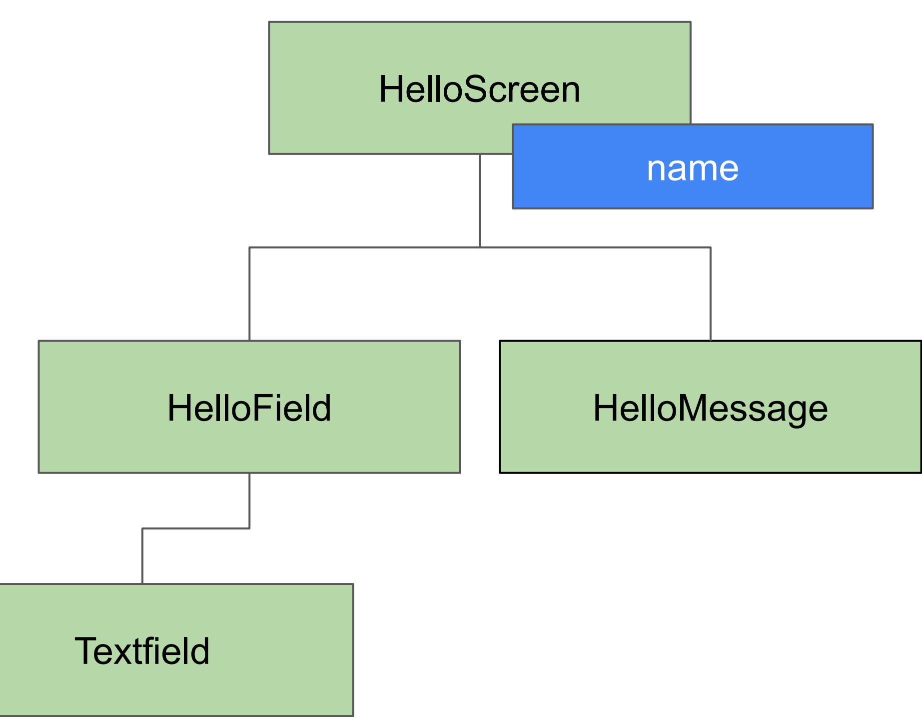
Logikk i en app kan være enten businesslogikk eller UI-logikk

- **UI-logikk** relateres til *hvordan å vise* tilstandsendringer i skjermen.
  - f.eks. navigasjonslogikk eller å vise snackbars
- **Businesslogikk** er *hva man skal gjøre* ved tilstandsendring
  - F.eks. å gjennomføre betaling eller å lagre brukerinstillinger
  - Denne logikken er normalt sett plassert i business eller datalag, aldri i brukergrensesnittet.



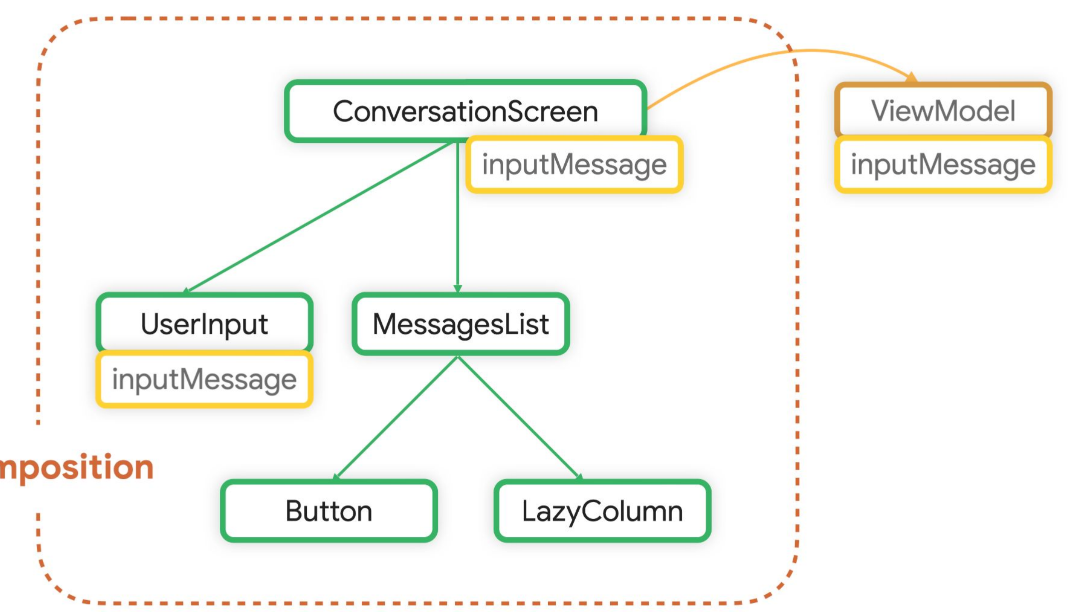
State hoisting

- Vi kan flytte tilstand ut av en composable med bruk av state hoisting
- Hvor vi løfter tilstanden til avhenger av om det er logikken til *brukergrensesnittet* eller *business-logikken* som trenger det.
- Du burde heise UI state til den “**laveste felles forelderen**” mellom alle composables som skal lese og skrive den
- Tilstand burde ligge nærmest der det blir “consumed”.
- Den som eier tilstanden burde eksponere “immutable state” og events for å modifisere tilstanden



State hoisting (forts.)

- Den laveste felles forelderen kan også være utenfor komposisjonen
- Dersom businesslogikk er involvert ønsker vi å flytte tilstanden ut av komposisjonen inn i en State Holder som f.eks. en ViewModel



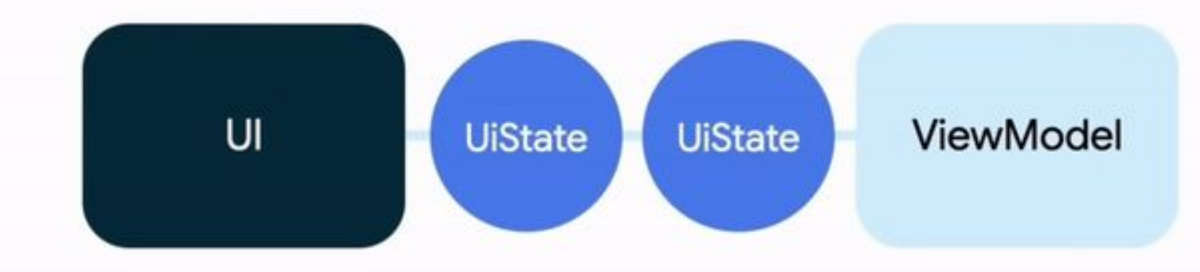
ViewModel

- Android Architecture Component
- Gir tilgang til business-logikk
- Eksponerer data til brukergrensesnittet gjennom en enkelt “property” kalt `uiState`.
- Ved flere urelaterte deler kan den inneholde flere uiStates

```kotlin
data class PokemonUIState(
   val pokemons: List<Pokemon>,
   val caughtPokemons: Set<Int>
)

class PokemonViewModel : ViewModel() {
      val pokemonUIState: StateFlow<PokemonUIState> = …

      …

      fun catchPokemon(id: Int){
            …
      }
}
```

ViewModel og UI state

- ViewModels eksponerer UIState
- Brukergrensesnittet konsumerer UIState og oppdaterer seg deretter

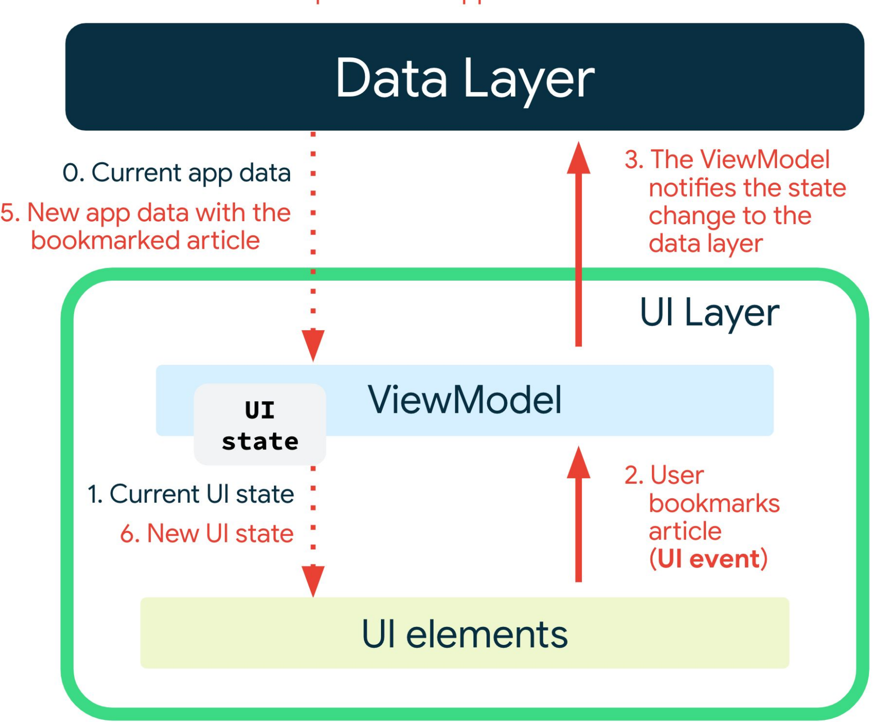


ViewModel

```kotlin
@Composable
fun PokemonScreen(pokemonViewModel: PokemonViewModel = viewModel()) {
    val pokemonUiState by pokemonViewModel.pokemonUIState.collectAsState()
    …
}
```

- Brukes på skjermnivå
- Lar deg persistere brukergrensesnittets tilstand
- Er ikke lagret som en del av komposisjonen
- De tilbys av rammeverket og er i skopet av en Activity, navigasjons-graf, eller en destinasjon i en navigasjons-graf
- Blir "cached" når destinasjonen er i backstacken

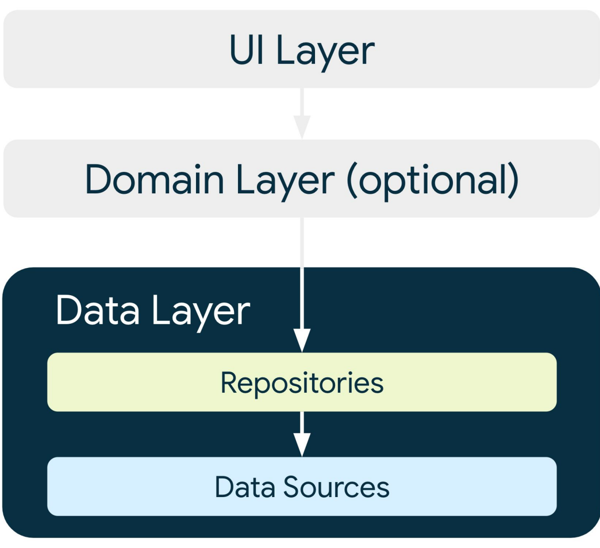
Flyt “mellom lagene”



Datalaget

Datalaget er bygd opp av *repositories* som hver kan inneholde ingen til mange *datakilder*.

- Hver datakilde burde ha ansvar for å kun jobbe med en kilde av data. Datakilder er broen imellom applikasjonen og systemet for data-operasjoner
    - F.eks. en fil
    - En nettverkskilde
    - Lokal database
- Operasjoner i datalaget skal være main-safe. Vi sier at en funksjon er main-safe hvis den ikke blokkerer at UI-et oppdaterer seg.



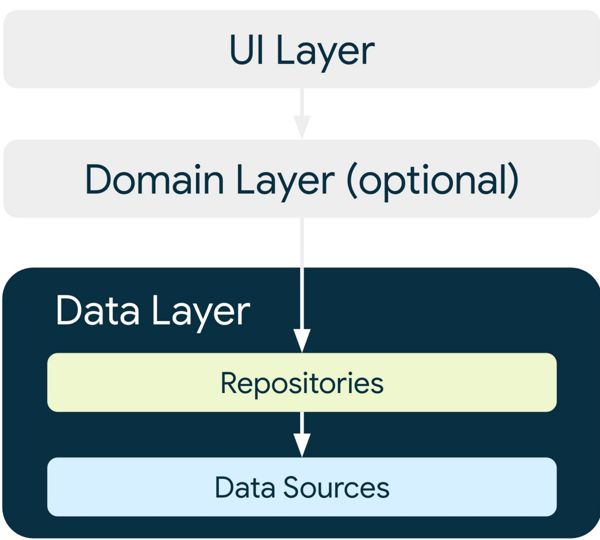
https://developer.android.com/topic/architecture/data-layer

Repositories

Repository klasser er ansvarlige for:

- Eksponere data til resten av appen.
- Sentralisere endringer i data.
- Håndtere konflikter mellom flere kilder av data
- Abstrahere bort datakildene fra resten av appen.
- Inneholde business-logikk.

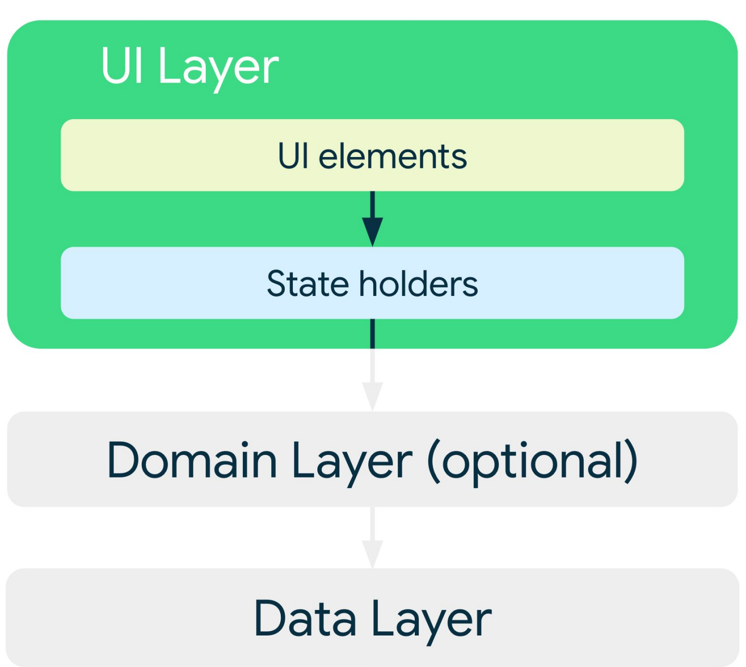
Skiller ofte repositories på hva slags data den inneholder, f.eks. MoviesRepository og PaymentsRepository



ViewModels og datalaget

ViewModel-en interagerer med data eller domenelaget ved bruk av:

- `suspend` funksjoner for å gjøre handlinger ved bruk av [`viewModelScope`]
- Kotlin flows for å motta applikasjonsdata



Pause! ☕

- Ta gjerne en kikk på pre-koden til demoen
- Fyll ut teaminnmeldings-skjema dersom du ikke har gjort det enda - se emnesiden, eller scan QR-koden;

Menti: **3397 5892**

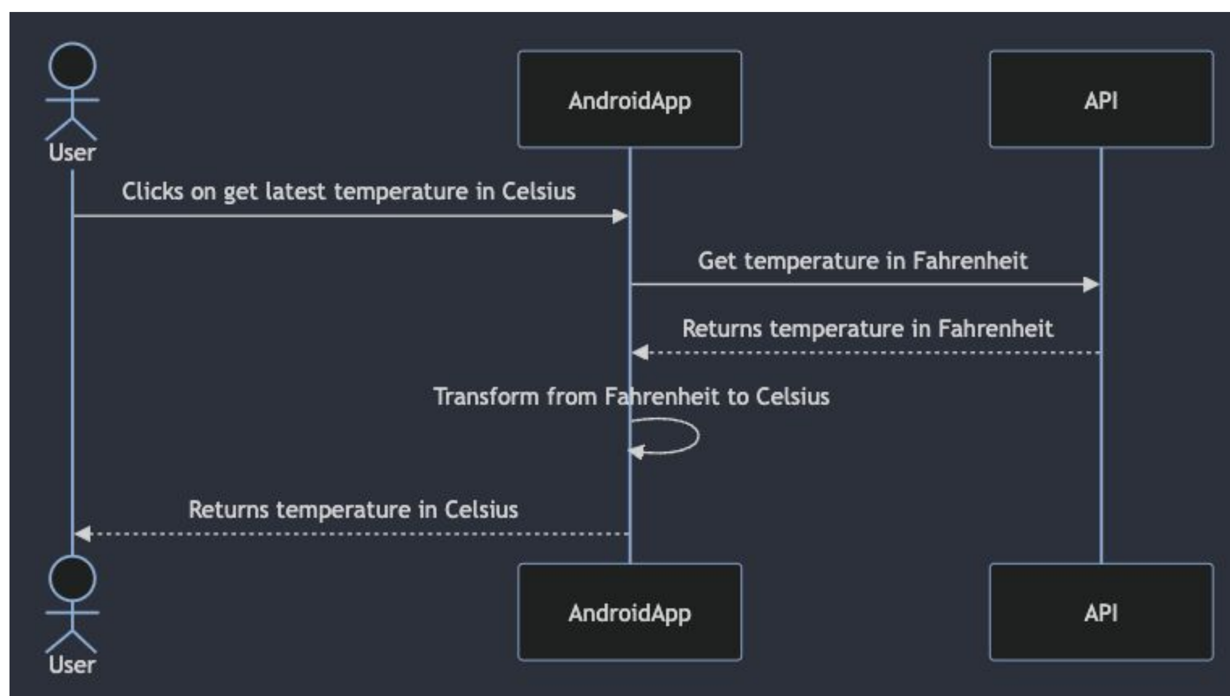
Samtidighet og parallellitet

Menti: **3397 5892**

Hvorfor samtidighet og parallellitet?

Se for deg:

En bruker interagerer med en AndroidApp som henter data fra et API og gjør en beregning.

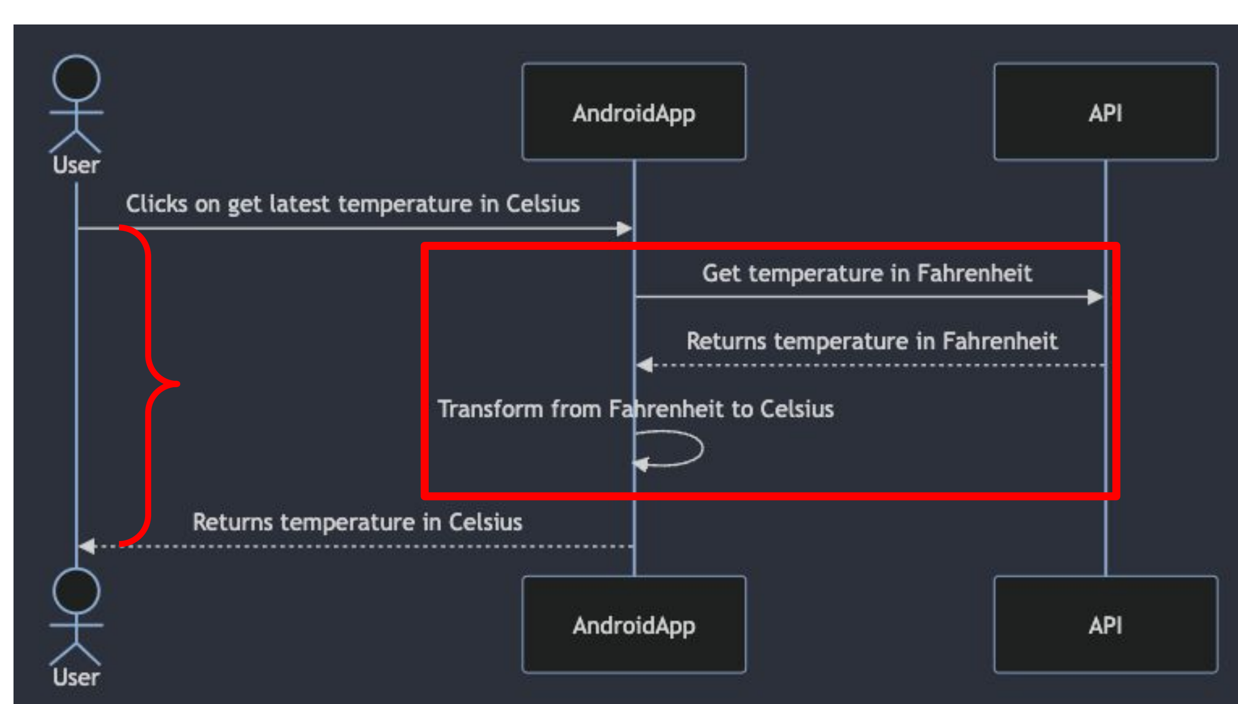


Hvorfor samtidighet og parallellitet?

Se for deg:

En bruker interagerer med en AndroidApp som henter data fra et API og gjør en beregning.

Vi ønsker IKKE at appen skal "henge" når appen gjør annet i bakgrunnen



Hva betyr samtidighet og parallellitet?

- **Samtidighet**
- På engelsk: *Concurrency*
- Samtidighet handler om å **håndtere** flere ting på en gang
  - Multitasking: til en hver tid gjør du kun en ting, men man har mange ting man må håndtere samtidig.

- **Parallellitet**
- På engelsk: *Parallelism*
- Parallellitet handler om å **gjøre** flere ting på en gang.
  - F.eks.: Utnytte flere prosesserings-enheter for å gjøre mindre oppgaver på likt, som resulterer i en raskere kjøretid.

- **Samtidighet != Parallellitet**

Synkron / asynkron

- **synkron**
  - Eksekvering skjer sekvensielt, dvs. en ting om gangen, etter hverandre, i en bestemt rekkefølge.
  - “blocking”

- **asynkron**
  - Eksekvering skjer ikke i en bestemt rekkefølge og flere oppgaver kan kjøre samtidig.
  - “non-blocking”

Coroutines

- Avbrytbar funksjon (suspendable)
- Funksjoner som tillater asynkron eksekvering av kode

- Coroutine builders
  - **launch** starter uten å forvente en returverdi
  - **async** starter men gir tilbake en lovnad om at det til slutt skal bli en verdi

https://kotlinlang.org/docs/coroutines-overview.html

CoroutineScope

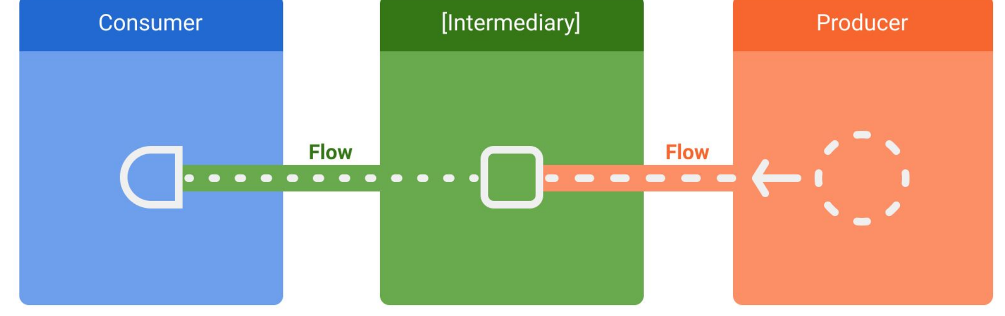
- Et interface som definerer et skop for Couroutines
- Du kan kun starte Couroutines fra et spesifikt CoroutineScope
- **launch** og **async** er kun definert inne i et CoroutineScope
- ViewModels tilbyr et eget CoroutineScope *viewModelScope*

Kotlin - Flows

- Konseptuelt en strøm av data som kan bli beregnet asynkront.

Involverer:

- En produsent av data
- Mellommenn
- Forbruker

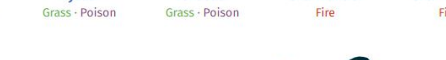


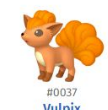
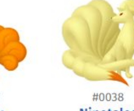
https://developer.android.com/kotlin/flow

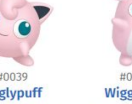
Demo

Menti: **3397 5892**

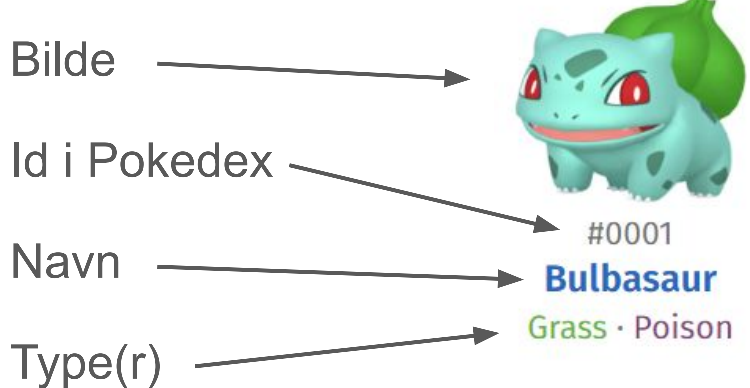
Demo - Pokemon app ("Mini-Pokedex")



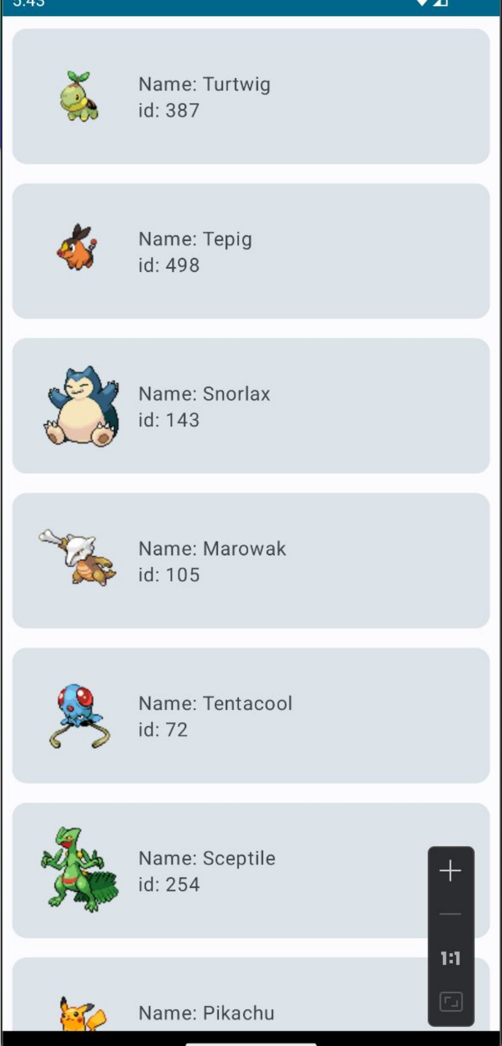





Demo - Pokemon app ("Mini-Pokedex")



...og MYE MER



ViewModel-anbefalinger

- **Bruk ViewModels på skjermnivå**. Altså ikke send Viewmodel-instanser nedover til andre composable-funksjoner og ikke bruk det i gjenbrukbare ui-komponenter. Der benytter du deg av en plain State holder.
- **ViewModels burde være uavhengig av Android sin lifecycle**. ViewModels burde ikke ha en referanse til noen Lifecycle-relaterte typer. F.eks. Context
- **Bruk Coroutines og Flows.**
- **Eksponér en UI state.**

https://developer.android.com/topic/architecture/recommendations

Hente data fra Web-API i en datakilde

- Trenger en http-klient
- Ktor client er en variant av det
  - Engine: Android eller okHttp
  - Serializer: kotlinx.serialization eller gson
- Et vanlig dataformat er JSON-formatet, dette kan vi ved hjelp av en serializer (serialiseringsmodul) gjøre om til Kotlin-objekter
- Mer i forelesning om API!