

IN2000

Software Engineering med prosjektarbeid

Introduksjon til Android og Kotlin

Menti: 3313 5425

Steffen Almås

Først...litt praktisk!

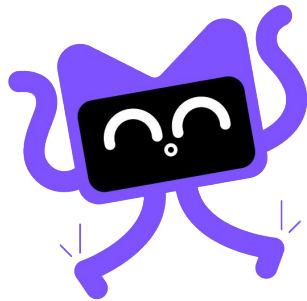
- Oblig 1 er nå ute
- Teaminnmelding
 - Skjema - husk å levere før fristen til oblig 1
 - Hvordan settes teamene sammen?
- Kickoff fredag 27. februar

Spørsmål - Menti

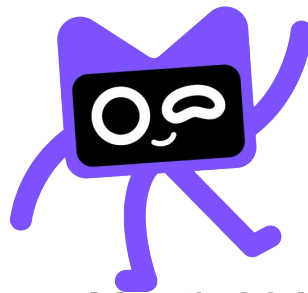
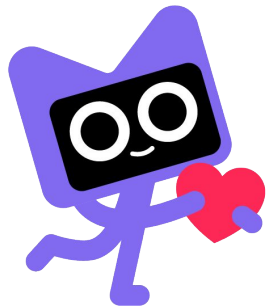
- Spørsmål underveis?
- Menti: **3313 5425**



Or use QR code



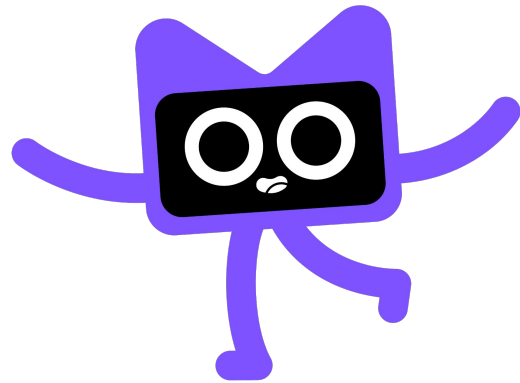
 **Kotlin**



Menti: 3313 5425

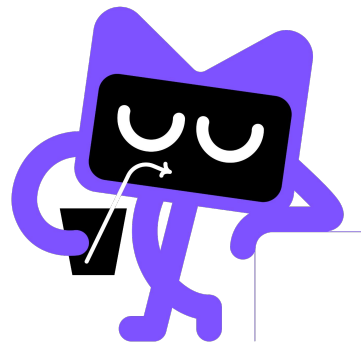
Forrige forelesning...

- Variabler
- Mutable / Immutable
- String templates
- Deklarering av funksjoner
- Interoperabilitet med Java
- Kotlin “ideomatiske” prinsipper



I dag!

- Kontrollflyt
- Klasser og objekter
- Kotlin's klassehierarki
- Null safety
- Samtidighet og parallellitet
 - Coroutines



Kontrollflyt

if/else

```
fun main() {  
    println(maxOf(3, 4))  
}  
  
fun maxOf(a: Int, b: Int): Int {  
    // Should return highest number  
}
```

if/else (forts.)

```
fun main() {  
    println(maxOf(3, 4))  
}  
  
fun maxOf(a: Int, b: Int): Int {  
    if (a > b) {  
        return a  
    } else {  
        return b  
    }  
}
```

if/else (forts.)

```
fun main() {  
    println(maxOf(3, 4))  
}  
  
fun maxOf(a: Int, b: Int): Int {  
    return if (a > b) {  
        a  
    } else {  
        b  
    }  
}
```

if/else (forts.)

```
fun main() {  
    println(maxOf(3, 4))  
}  
  
fun maxOf(a: Int, b: Int): Int {  
    if (a > b) {  
        return a  
    } else {  
        return b  
    }  
}
```



```
fun main() {  
    println(maxOf(3, 4))  
}  
  
fun maxOf(a: Int, b: Int): Int {  
    return if (a > b) {  
        a  
    } else {  
        b  
    }  
}
```

if/else - Expression function

```
fun main() {  
    println(maxOf(3, 4))  
}  
  
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

when

```
fun main() {  
    numToWord(5)  
}  
  
fun numToWord(x: Int) {  
    when(x) {  
        0 -> print("Zero")  
        1 -> print("One")  
        2 -> print("Two")  
        3 -> print("Three")  
        4 -> print("Four")  
        5 -> print("Five")  
        6 -> print("Six")  
        7 -> print("Seven")  
        8 -> print("Eight")  
        9 -> print("Nine")  
        else -> {  
            print("What?")  
        }  
    }  
}
```

for

```
val items = listOf("Alpacas", "are", "cool")

for (item in items) {
    println(item)
}

for ((index, value) in items.withIndex()) {
    println("$index : $value")
}
```

for

```
val items = listOf("Alpacas", "are", "cool")

for (item in items) {
    println(item)
}

for ((index, value) in items.withIndex()) {
    println("$index : $value")
}
```

for (forts.)

```
val items = listOf("Alpacas", "are", "cool")

for (item in items) {
    println(item)
}

for ((index, value) in items.withIndex()) {
    println("$index : $value")
}
```

forEach

```
val items = listOf("Alpacas", "are", "cool")  
  
items.forEach { println(it) }
```

OUTPUT

```
>> Alpacas  
>> are  
>> cool
```

for (forts.)

```
for (i in 1..3) {  
    println(i)  
}
```

OUTPUT

```
>> 1  
>> 2  
>> 3
```

```
for (i in 6 downTo 0 step 2) {  
    println(i)  
}
```

OUTPUT

```
>> 6  
>> 4  
>> 2  
>> 0
```

while

```
val items = listOf("IN2000", "is", "cool")
var i = 0

while (i < items.size) {
    println("item at $i is ${items[i]}")
    i++
}
```

Demo - Kontrollflyt og loops

- Oppgave
 - Finn summen av partall i listen
- Løsningsforslag

Klasser og objekter

Kotlin - class

- Enkel klasse

```
class Person {  
    val age = 10  
}  
  
fun main() {  
    val person = Person()  
    println(person.age)  
}  
  
>> 10
```

Kotlin - primary constructor

```
class Person(age: Int) {  
    val age: Int = age  
}  
  
fun main() {  
    val person = Person(10)  
    println(person.age)  
}
```

Kotlin - primary constructor (with properties)

- Slipper å sette variablene -
det gjøres automatisk

```
class Person(val age: Int)

fun main() {
    val person = Person(10)
    println(person.age)
}
```

Kotlin - secondary constructor

- “constructor”- keyword

```
class Person {  
    val age: Int  
    constructor(age: Int) {  
        this.age = age  
    }  
}  
  
fun main() {  
    val person = Person(10)  
    println(person.age)  
}
```

Recap - Konstruktører - oppsummert

secondary constructor	primary constructor	primary constructor with properties
<pre>class Person { val age: Int constructor(age: Int) { this.age = age } }</pre>	<pre>class Person(age: Int) { val age: Int = age }</pre>	<pre>class Person(val age: Int)</pre>

- Alle brukes på samme måte:

```
fun main() {  
    val person = Person(10)  
    println(person.age)  
}
```

Kotlin - init

- Innhold i init-blokk blir kalt ved oppretting av objektet

```
class Person(val age: Int) {  
    init {  
        println("Creating person with age: $age")  
    }  
}  
  
fun main() {  
    val person = Person(10)  
    println(person.age)  
}
```

Kotlin - Rekkefølge konstruktører og init

- Rekkefølge på initialisering:
 1. primary constructor
 2. `init`
 3. secondary constructor

Kotlin - data class

- Brukes for å representere data
- Får en del automatisk
- [Eksempel class vs data class](#)

```
data class Person(val age: Int)

fun main() {
    val person = Person(10)
    println(person.age)
}
```

Kotlin - Arv

: brukes for både **extends** og **implements**

```
open class Car(val plateNumber: String)

class Elcar(plateNumber: String, val batteryCapacity: Int) : Car(plateNumber)

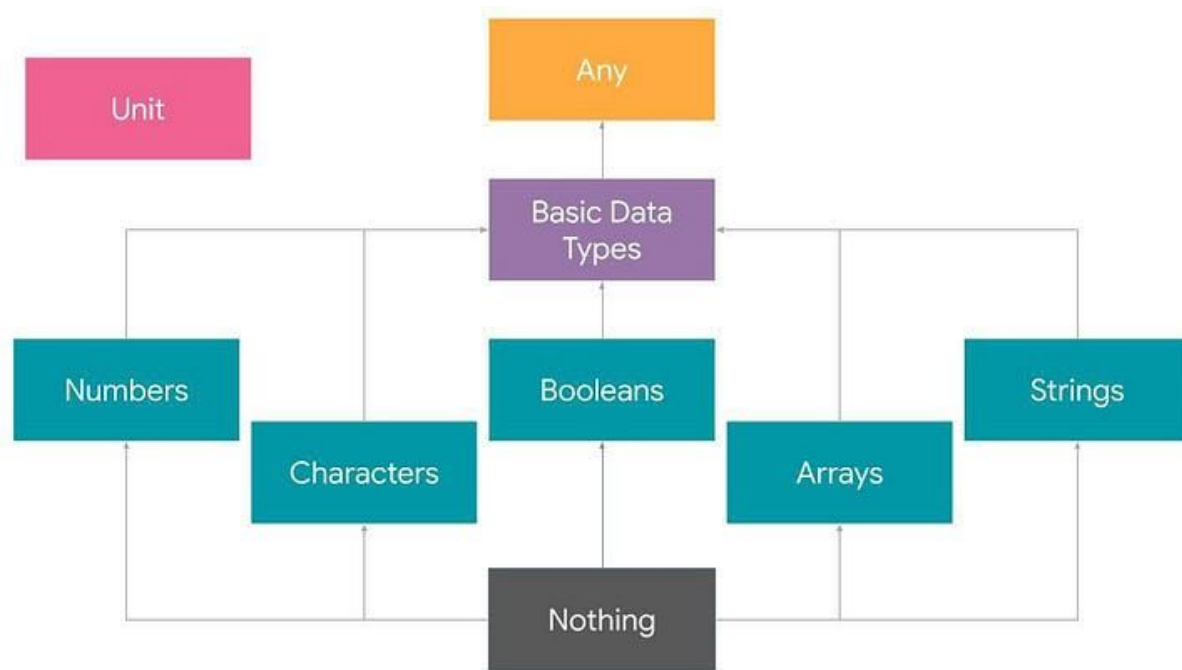
fun main() {
    val elCar = Elcar("IN2000", 10000)
    println(elCar.plateNumber)
    println(elCar.batteryCapacity)
}
```

Kotlins klassehierarki

Klasser og objekter

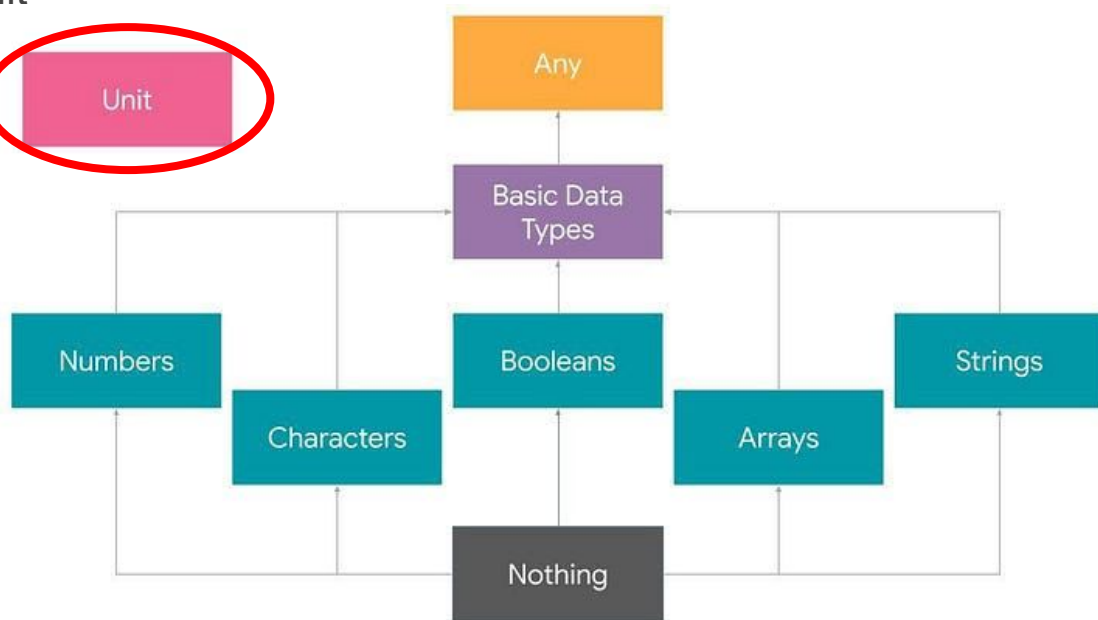
- “Er variabler typer eller objekter?”
- **Variabler er objekter**
 - Og skal se mer på det nå!

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



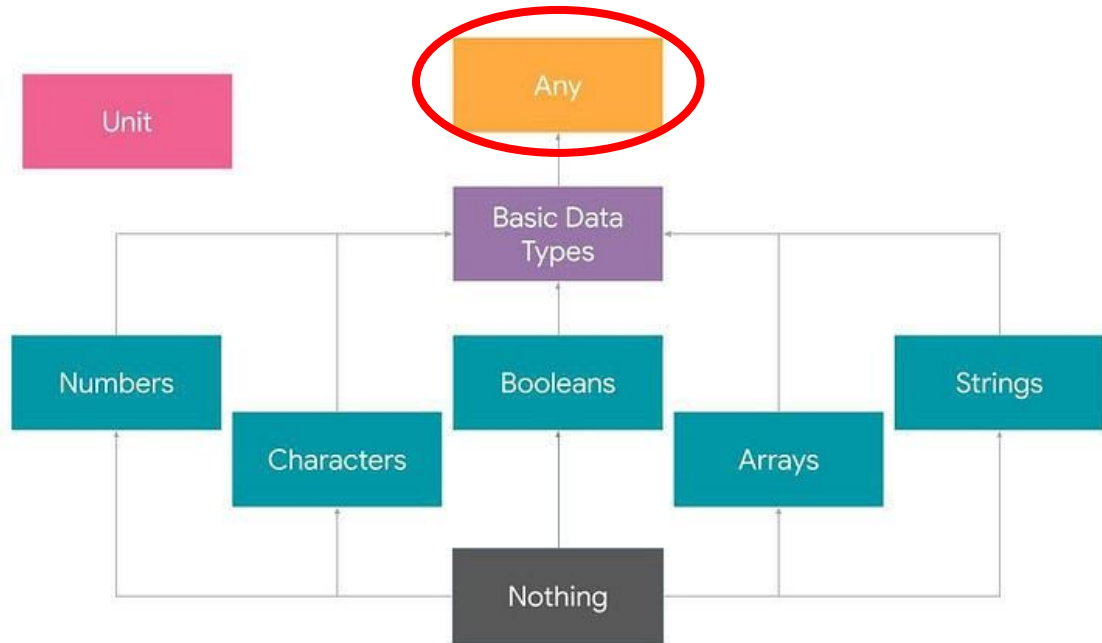
```
fun main() {  
    printHello()  
}  
  
fun printHello(): Unit {  
    println("Hello World")  
}
```

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 - 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
```

```
}
```

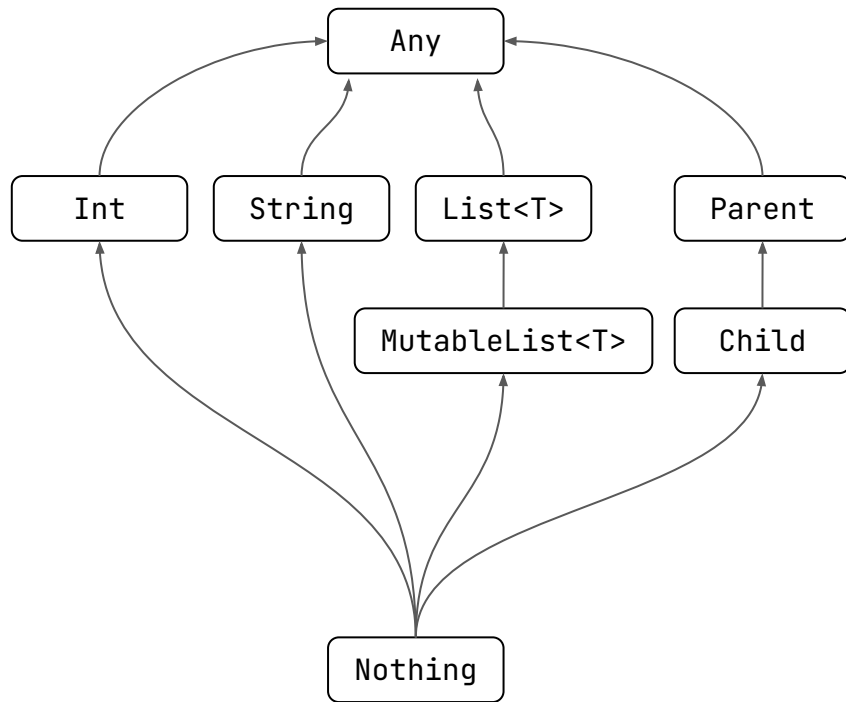
```
fun main() {
```

```
    val today = Day.MONDAY
```

```
    println(if (today == Day.SATURDAY || today == Day.SUNDAY) "Weekend!" else "Weekday...")
```

```
}
```

Kotlins klassehierarki (uten Nullable typer)



Null safety

Null Safety

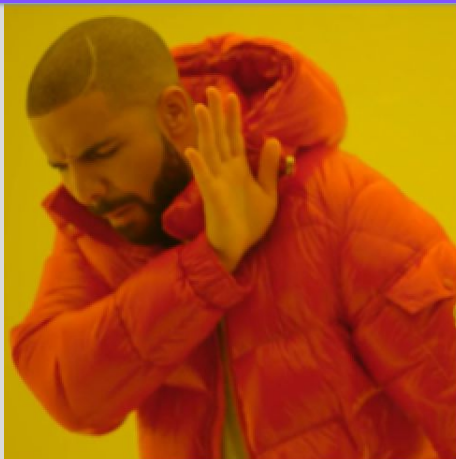


" I call it my
billion-dollar mistake.
It was the invention of
the null reference in
1965 "
- Tony Hoare -

Null Safety

- Hva er null?
 - En peker til “ingenting”
- NullPointerException
- Kotlin har innebygd Null Safety
- **Nullable types**

Null Safety - Nullable types



```
val x: String = null
```



```
val x: String? = null
```

Nullable types

- `String` $\rightarrow$ `String?`
- `Int` $\rightarrow$ `Int?`
- `Double` $\rightarrow$ `Double?`
- `Boolean` $\rightarrow$ `Boolean?`

Null Safety - Hva skjer?

```
val x: String? = "abc"
```

```
println(x.length)
```

```
val y: String? = null
```

```
println(y.length)
```

Null Safety - Hva skjer?

```
val x: String? = "abc"
```

```
println(x.length)
```



```
val y: String? = null
```

```
println(y.length)
```



Null Safety - Non null assertion operator

- !! bruk hvis du har kontroll

```
val name: String? = "IN2000"  
val length = name!!.length  
  
println(length)
```

- Men hva skjer hvis name faktisk er **null**?

Null Safety - Safe call operator

- ?. er trygg å bruke

```
val name: String? = "IN2000"  
val length = name?.length  
  
println(length)
```

- Men hva skjer hvis name faktisk er **null**?

Null Safety - Elvis operator

- Elvis operator = `?:` (ser ut som Elvis)
- Hvis verdi til venstre er null, så brukes verdi til høyre
- Som regel tryggeste alternativ

```
val name: String? = getName()  
val default = "IN2000"  
  
val finalName = name ?: default
```

Null Safety - Smart casting

- Smart casting = Kotlin-kompilatoren sjekker “is-checker” og eksplisitt casts, og setter inn trygge kall (safe calls) automatisk!

```
val x: String? = null

if (x != null) {
    println(x.length)
}
```

Null Safety - Smart casting (forts.)

- Smart casting = Kotlin-kompilatoren sjekker “is-checker” og eksplisitt casts, og setter inn trygge kall (safe calls) automatisk!
- Kan også “inlines”

```
val x: String? = null

if (x != null && x.length > 2) {
    ...
}
```

Null Safety - Smart cast med *requireNotNull*

```
fun checkSize(x: String?): Int {  
    requireNotNull(x) { "Require input to be non-null" }  
    return x.length  
}  
  
fun main() {  
    println(checkSize("Hello")) // -> returns 5  
    println(checkSize(null))    // -> throws IllegalArgumentException  
}
```

Null Safety - casting feature ✨

Kotlin 2.0

```
class Cat {  
    fun purr() {  
        println("Purr purr")  
    }  
}  
  
fun petAnimal(animal: Any) {  
    if (animal is Cat) {  
        animal.purr()  
    }  
}
```

Null Safety - casting feature ✨

Kotlin 2.0

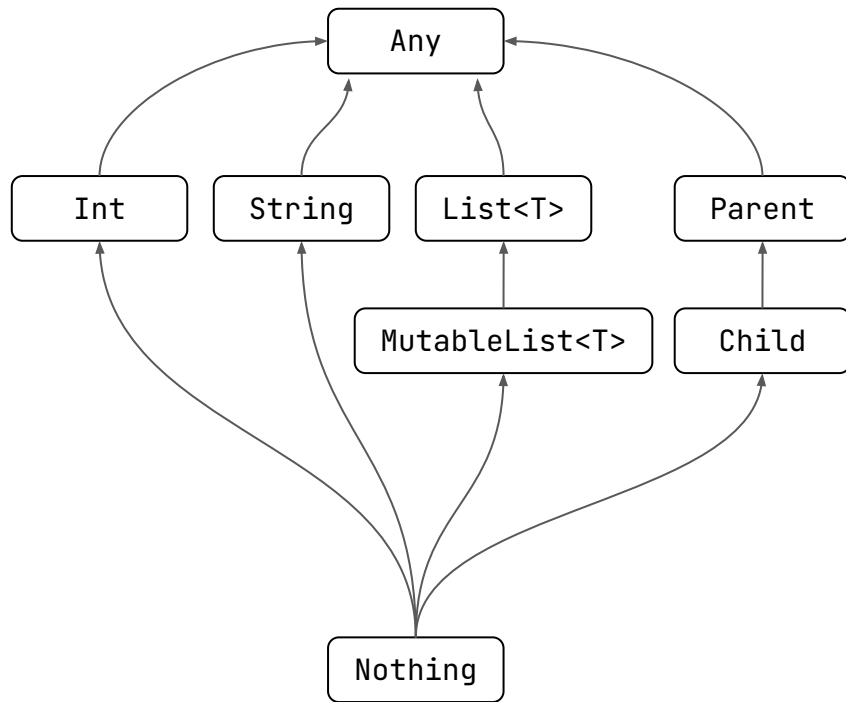
```
class Cat {  
    fun purr() {  
        println("Purr purr")  
    }  
}  
  
fun petAnimal(animal: Any) {  
    val isCat = animal is Cat  
    if (isCat) {  
        animal.purr() // Error in Kotlin 1.x  
    }  
}
```

Null Safety - casting feature ✨

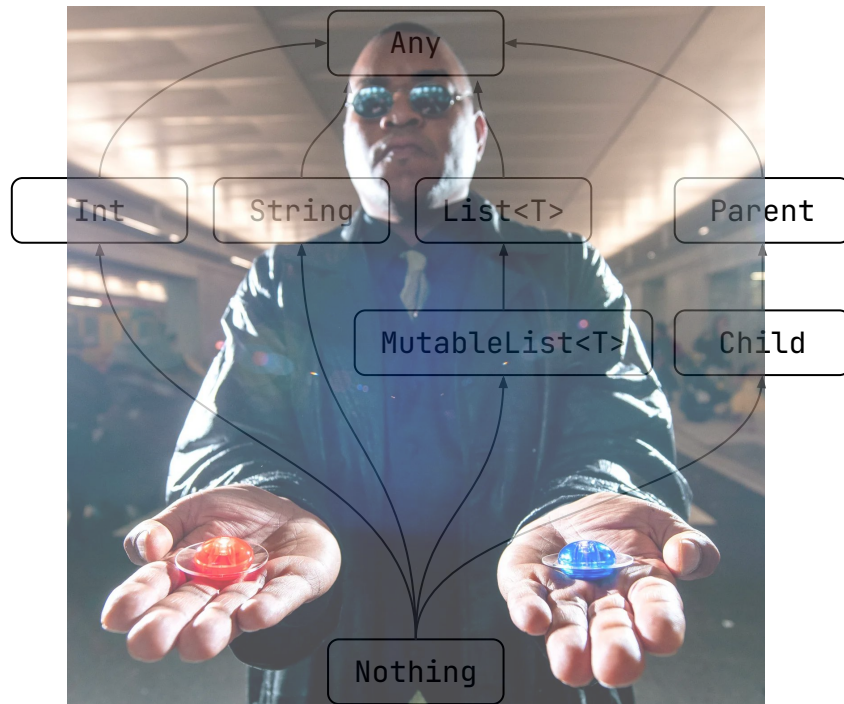
Kotlin 2.0

```
class Cat {  
    fun purr() {  
        println("Purr purr")  
    }  
}  
  
fun petAnimal(animal: Any) {  
    val isCat = animal is Cat  
    if (isCat) {  
        animal.purr() // OK in 2.0  
    }  
}
```

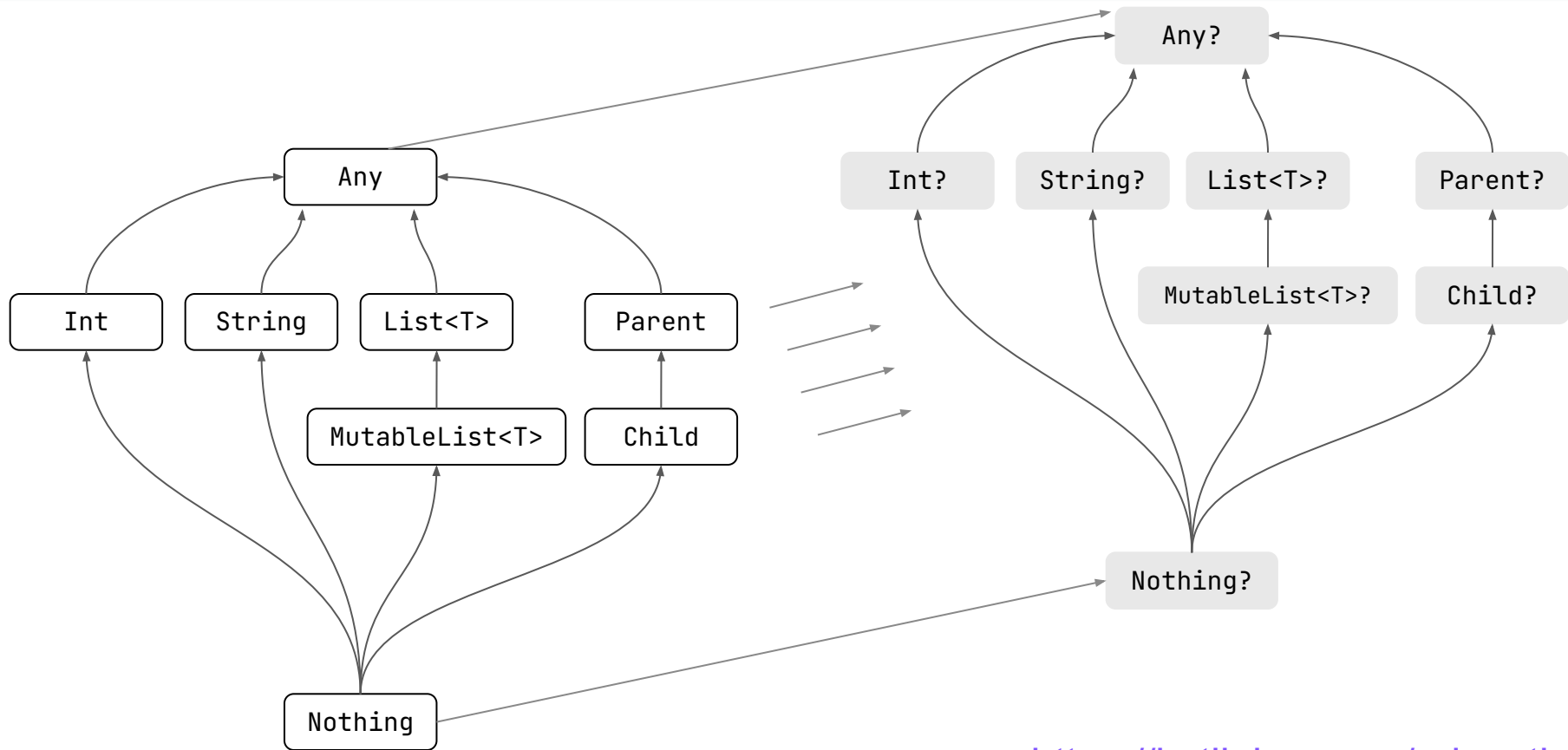
Nullable types, hva med Kotlins klassehierarki? 🤔



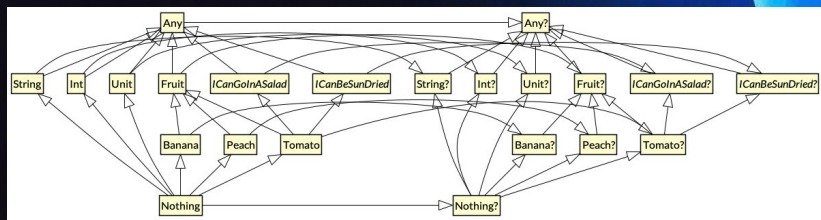
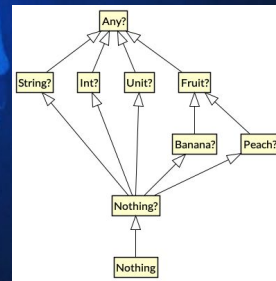
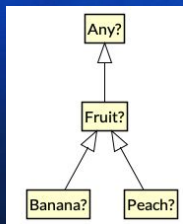
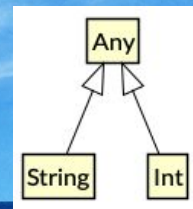
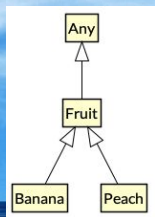
Nullable types, hva med Kotlins klassehierarki? 🤔



Kotlins fulle klassehierarki



Klasser og objekter - Kotlin klassehierarki, isfjell

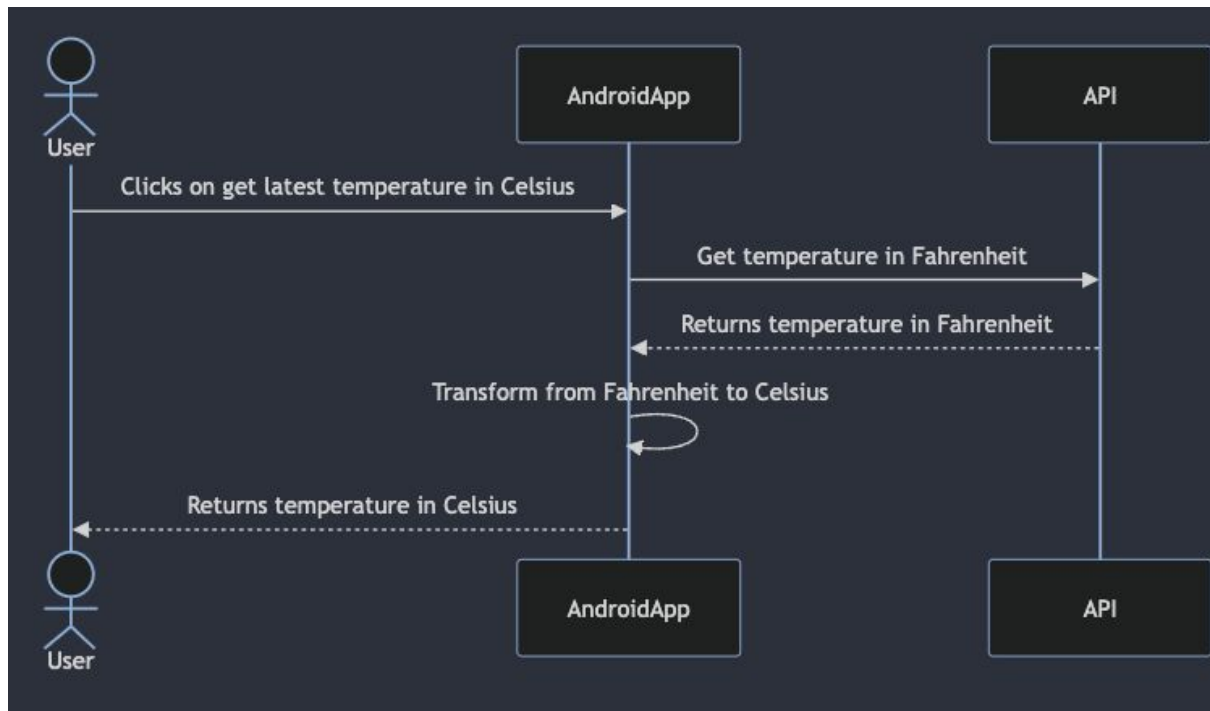


Samtidighet og parallellitet

Hvorfor samtidighet og parallellitet?

Se for deg:

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

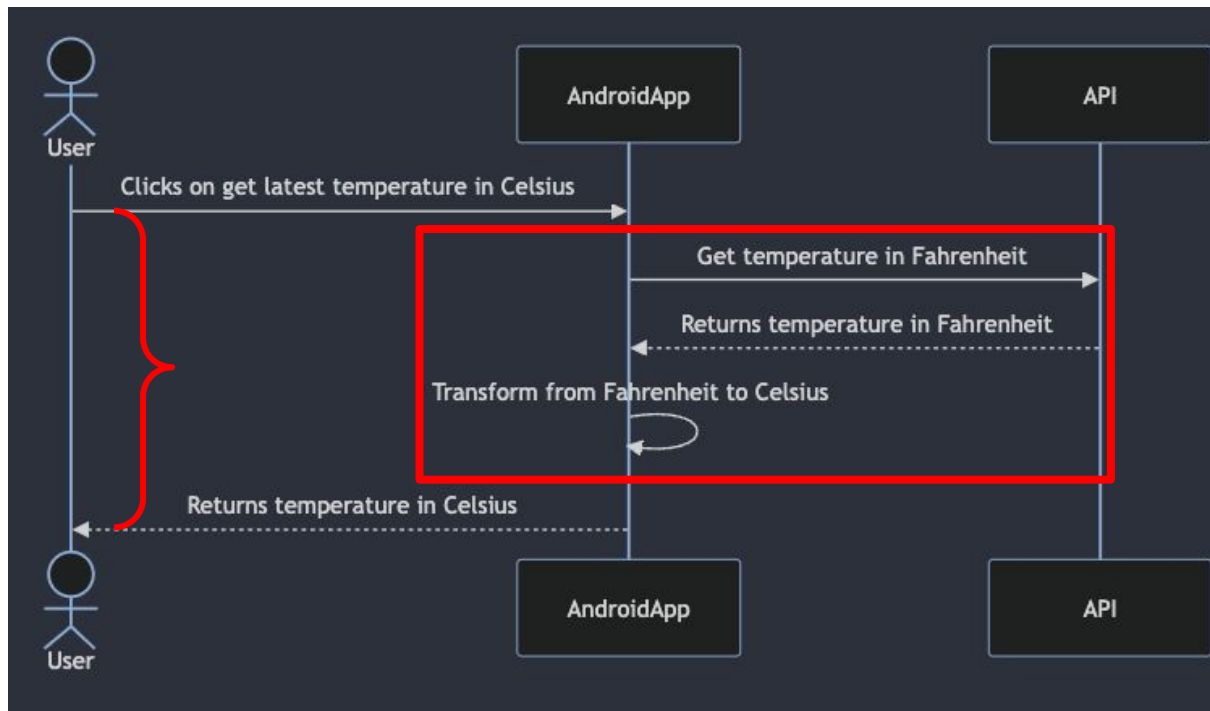


Hvorfor samtidighet og parallellitet?

Se for deg:

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

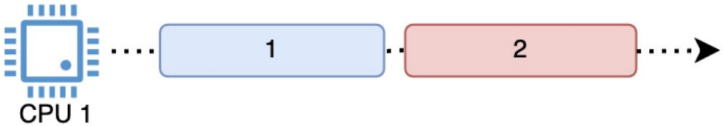
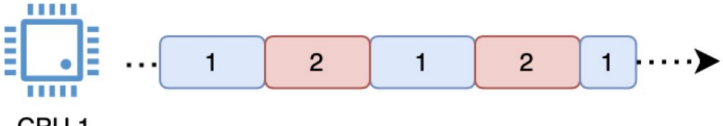
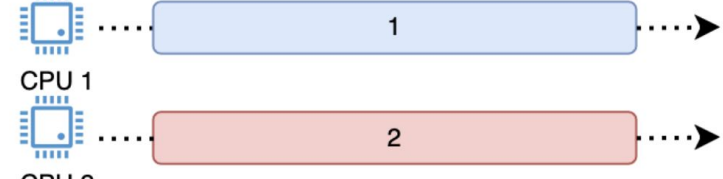
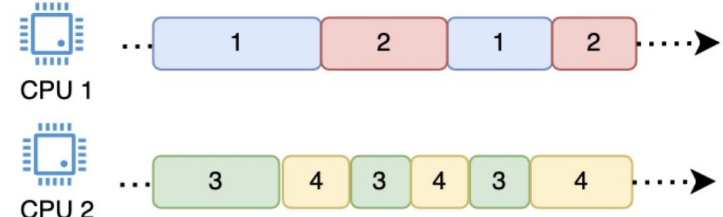
Vi ønsker IKKE at appen skal “henge” når oppgaver gjøres i bakgrunnen



Hva betyr samtidighet og parallellitet?

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

Hva betyr samtidighet og parallellitet? (forts.)

Ikke parallell og ikke samtidig	 CPU 1
Samtidig, men ikke parallell	 CPU 1
Parallell, men ikke samtidig	 CPU 1 CPU 2
Parallell og samtidig	 CPU 1 CPU 2

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



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](#)

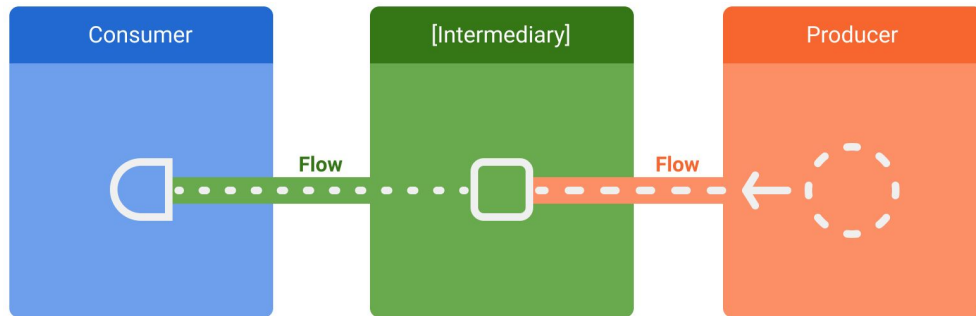


Kotlin - Flows

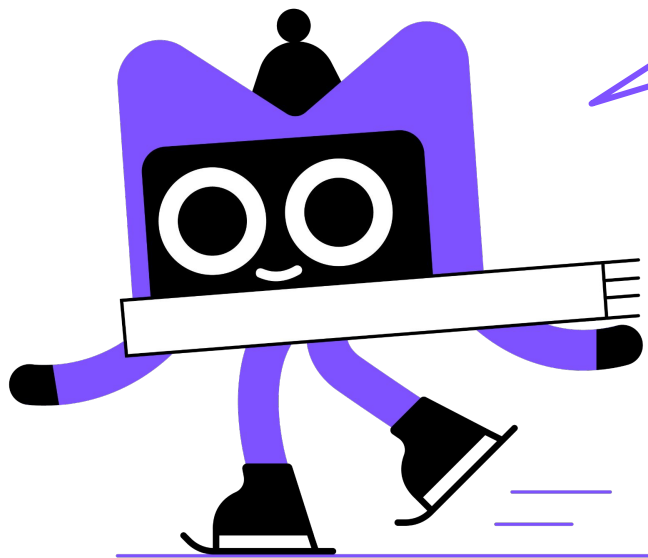
- Konseptuelt en strøm av data som kan bli beregnet asynkront.

Involverer:

- En produsent av data
- Mellommenn
- Forbruker

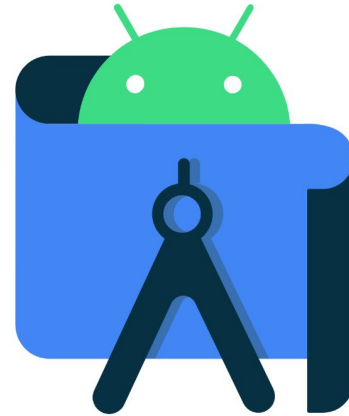


Kotlin - Funksjonell programmering



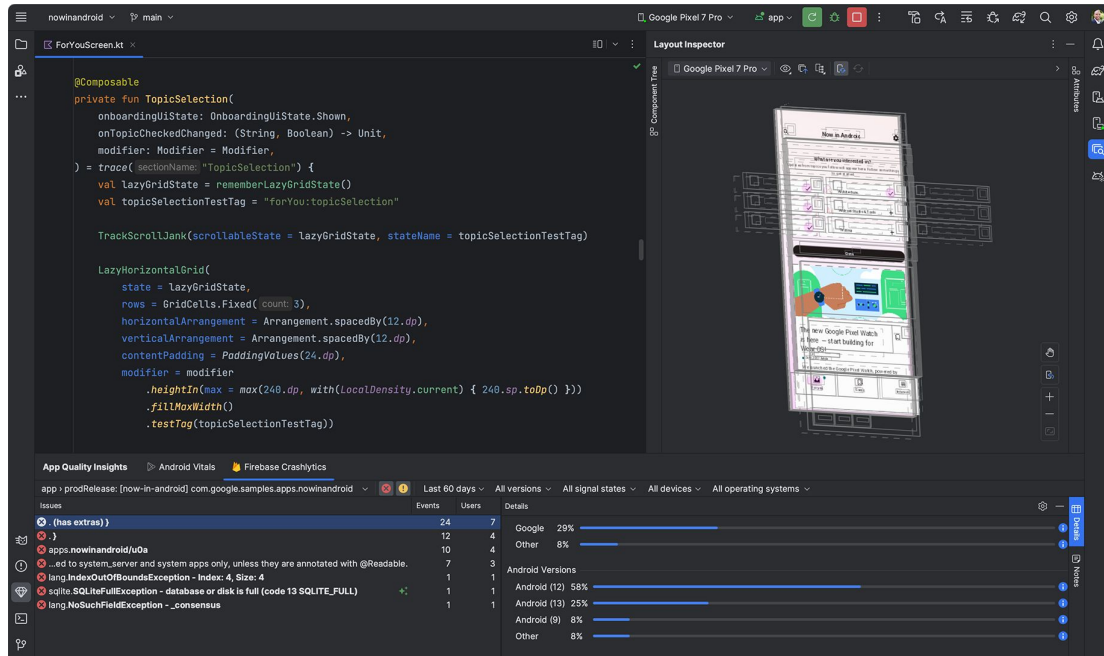
Om to uker!

android studio



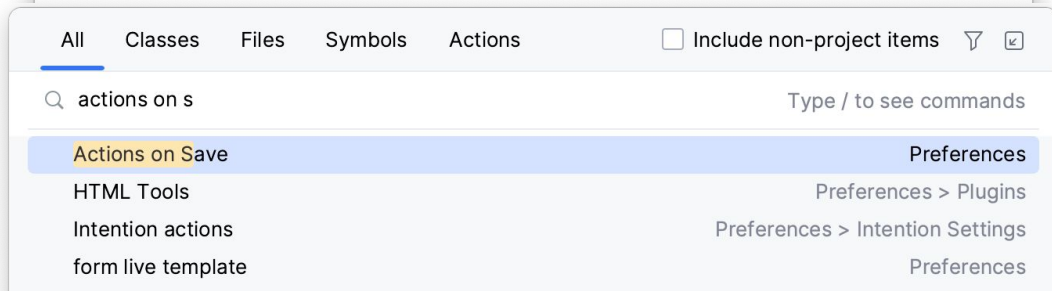
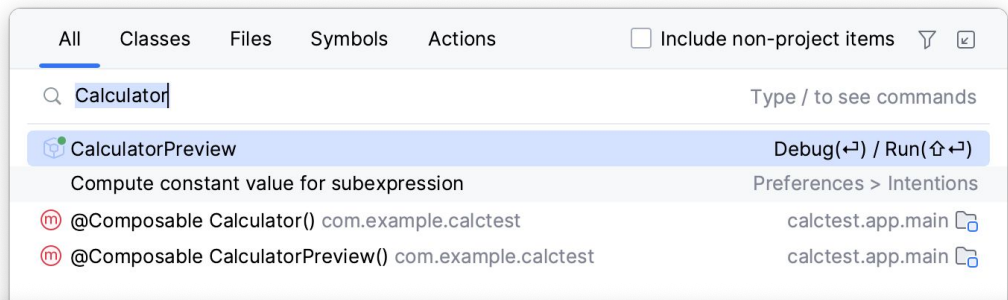
Bli bedre kjent med Android Studio

- Android Studio er basert på IntelliJ IDEA
- En kraftig IDE med mange muligheter
- Debugger
- Tooling for compose:
 - Previews
 - Layout editor



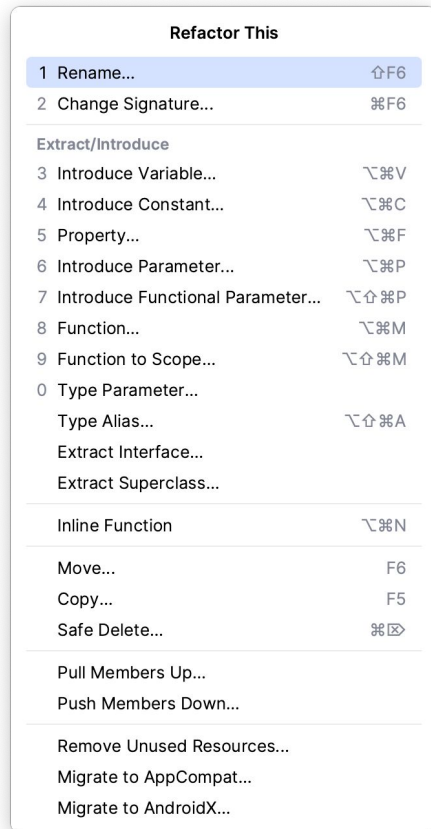
Hovedsøkbaren

- Double-shift for å få den opp
- Kan søke i blant annet
 - Innstillinger
 - Filer
 - Klasser, Symboler osv
 - Handler en kan gjøre



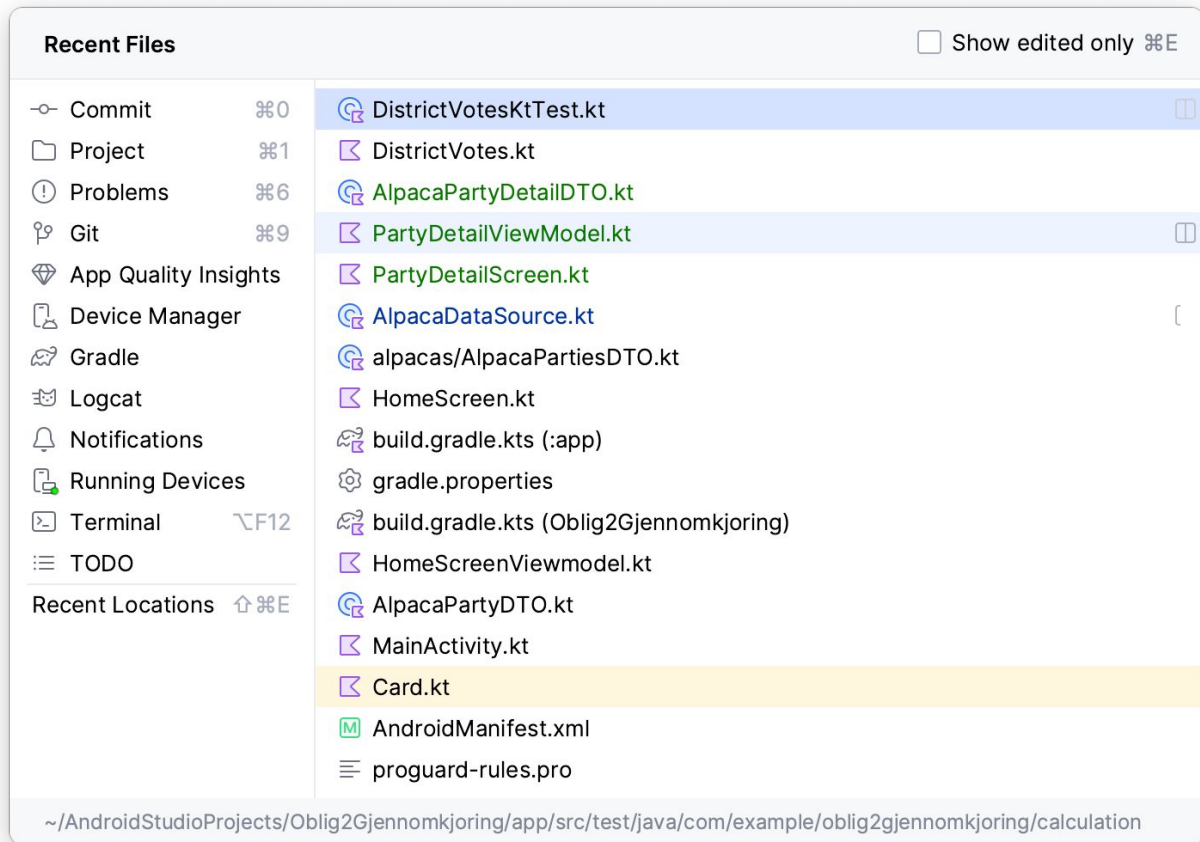
Refaktoring

- Masse innebygd i Android Studio
- Ctrl + T når markøren er på det du vil refaktorere
- Du kan angre hele refaktoringen med CMD + Z



Finne nylige filer

- Cmd + E



Andre nyttige snarveier

- **cmd + klikk** eller **cmd + B**, gå til definisjonen
- **option + enter**, vis konteksthandlinger, quick-fikser for feil og warnings, importer, og andre ting for å forbedre og optimalisere koden
- Se mer her:
<https://www.jetbrains.com/help/idea/mastering-keyboard-shortcuts.html>

Actions on save

- Hver gang en lagrer (cmd + S) vil oppgaven her utføres
- God hjelp med å holde koden ryddig

Tools > Actions on Save 			Reset	←	→
Action			Activated On		
<input checked="" data-bbox="189 644 218 666" type="checkbox"/>	Reformat code	All file types ▾	Any save		
<input checked="" data-bbox="189 751 218 773" type="checkbox"/>	Optimize imports	All file types ▾	Any save		

Gradle



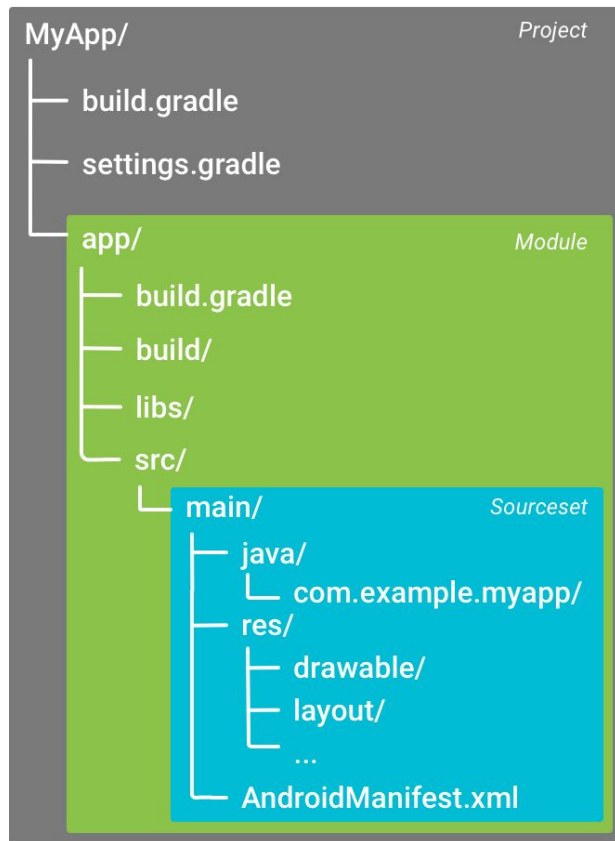
tool” som er designet for å kunne
vare.

r å gjøre ting som allerede er gjort
ell for raskere ytelse



Gradle - build.gradle

- build.gradle (**Project**)
 - Lokalisert i rotmappen til prosjektet
 - Spesifiserer konfigurasjoner som er felles for hele prosjektet
- build.gradle (**Module**)
 - Spesifisere konfigurasjon til moduler
 - Biblioteker legges her



Gradle - dependencies

- *implementation* – “Vanlig” avhengighet
- *testImplementation* - avhengighet for tester
- *androidTestImplementation* - avhengighet for tester som er avhengige av Android-funksjonalitet

Android Studio - alt ser riktig ut, men det funker ikke?

Nye avhengigheter:

- Gradle sync (hvis du nylig har lagt inn nye avhengigheter)
- Build Clean

Starter ikke i emulator:

- Wipe data på emulator
- Test en annen emulator

Feil i Android Studio:

- Invalidate Caches

Android Studio - Plugins

- Forbedre utvikleropplevelsen (og ha det gøy! ✨)

- Min favoritt:



Dracula Theme

↓ 2.4M ☆ 4.40 Zihan Ma

Install



Power Mode II

★★★★★

axaluss

Power Mode Based on the activate-power-mode atom plugin and forked from Baptiste Mesta on Github. Features Exploding falling

375,278 downloads

Free



Nyan Progress Bar

★★★★★

Dmitry Batkovich

Pretty progress bars with nyan cat for IntelliJ based IDEs.

1,240,875 downloads

Free



Anime Memes

★★★★★

Alex Keith Simons

Give your IDE more personality and have more fun programming with the Anime Meme IDE Integration! (AMII) Upon

69,665 downloads

Free



Jetpack Compose

Hva er Jetpack Compose?

- Anbefalte rammeverk for brukergrensesnitt i Android
- Lansert i 2019, første stabile versjon i 2021
- **Deklarativt UI-rammeverk**
 - Regenererer hele skjermen fra “scratch”
 - Compose velger hva som blir oppdatert til enhver tid for å ikke måtte generere hele skjermen på nytt



Jetpack Compose

- En komposisjon (Composition) beskriver utseende til appen
- En komposisjon er bygget opp av flere Composable-funksjoner (tenk på musikk)
- Composables er Immutable, dvs.
 - At den ikke kan endres etter at den er opprettet.
 - UI må oppdateres, dette kalles rekomponering (Recomposition)

Composable funksjoner

- Annoteres med `@Composable`
- PascalCase
- Har ingen returverdi
- Bør være idempotent (*ikke ha sideeffekter*)
- Kan bruke Kotlin som vanlig i funksjonen,
f.eks. bruke if/else for kontrollflyt
og velge hva som skal vises
basert på logikk

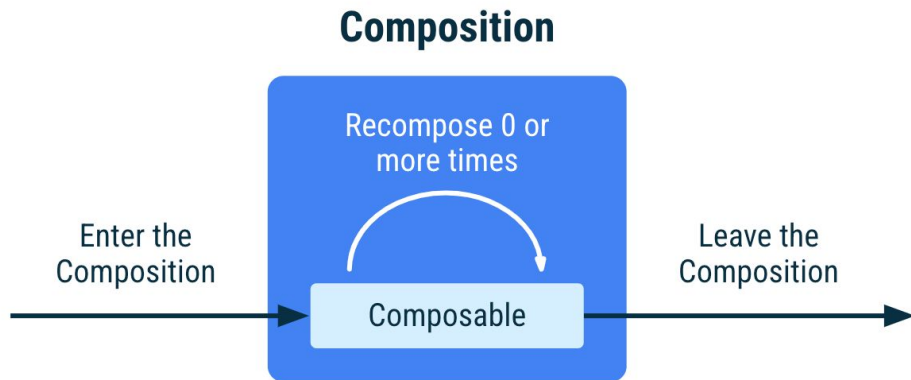
```
@Composable  
fun Greeting(name: String) {  
    Text("Hello $name")  
}
```

Compose - Noen begreper

Komposisjonen er beskrivelsen av brukergrensesnittet som blir bygd av Jetpack Compose når den kjører composables.

Initiell komposisjon er den første opprettelsen av komposisjonen

Rekomposisjon er det å kjøre en composable på nytt når informasjonen/input til funksjonen endres



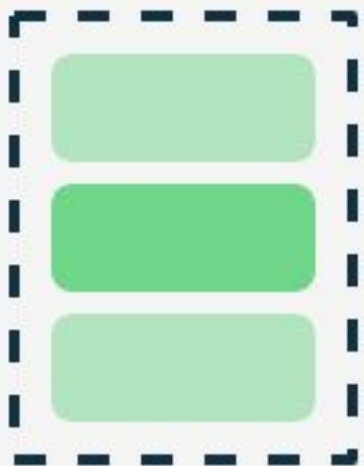
Jetpack Compose

- **Surface** er en container som representerer en del av UI-et hvor du kan endre på utseende - f.eks. bakgrunnsfarge
- **Modifier** - brukt for å dekorere, endre oppførselen til en composable eller å gi tilleggsinformasjon
 - Padding, margin, interaksjoner (f.eks. klikk), scrolling ol.
 - Semantisk informasjon
- **Preview** lar deg se hvordan appen ser ut uten å bygge hele appen
 - Markeres med **@Preview**-annotasjon

Android og Compose

- **onCreate** - inngangen til appen, kaller på funksjoner for å bygge UI
 - Kotlin-programmer er det main-funksjonen som er inngangen, i Android er det onCreate-funksjonen som har denne rollen
- **setContent** - brukes til å definere layouten basert på Composable-funksjoner
- **Composable-funksjoner** forteller kompilatoren at funksjonen brukes av Jetpack Compose for å generere UI-et (Brukergrensesnitt)
 - Tar input og generer det som vises på skjermen
 - Markeres med @Composable

Byggeblokkene; Column, Row, Box



Column



Row



Box

Størrelser

- device-independent pixels (dp)
 - Uavhengig av oppløsningen til skjermen
- scalable pixels (sp)
 - For tekst
 - Skalerer når en i innstillingene til mobilen justerer skriftstørrelsen

```
Box (  
    Modifier.padding(  
        10.dp, // Int  
        10f.dp, // Float  
        20.0.dp, // Double  
        10.dp  
    )  
)
```

Noen innebygde Modifiers

- Padding
- Size
- Offset
- Det finnes forskjellige modifiers for forskjellige composables!
 - Box: `matchParentSize`
- <https://developer.android.com/jetpack/compose/modifiers-list>

Preview-funksjoner

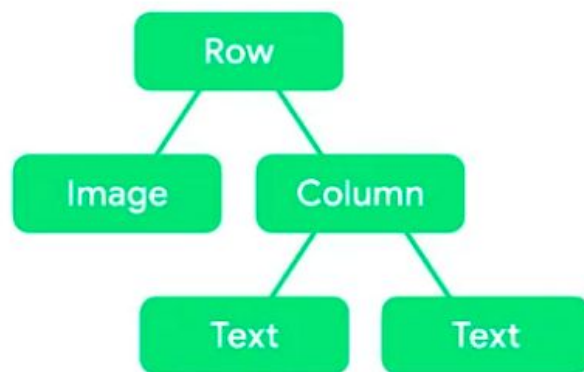
- Kan brukes til forhåndsvisning av funksjoner under “design”-fanen
- Kan også kjøres i emulator (knapp til venstre)
- I motsetning til funksjonene i setContent i MainActivity, så kjøres denne i “isolasjon”

@Preview

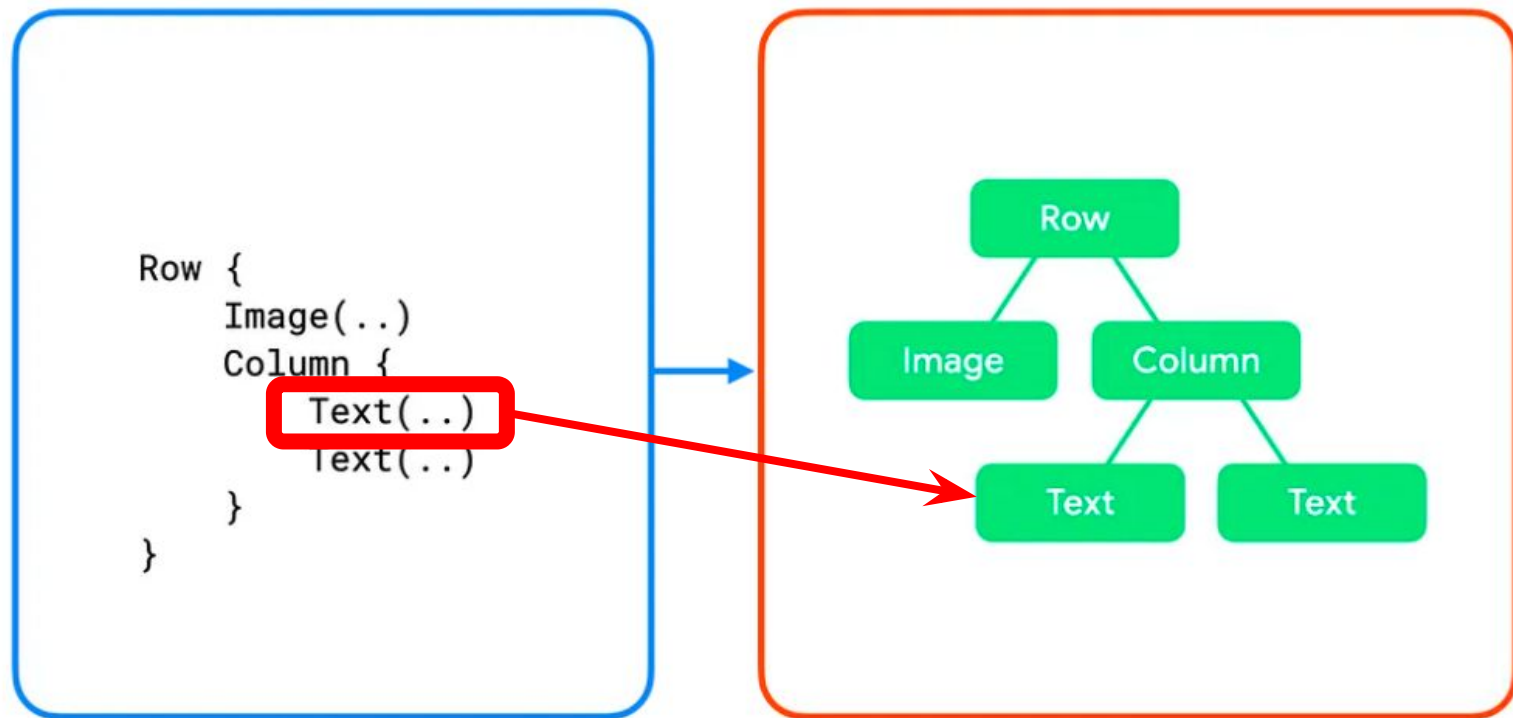


UI-treet

```
Row {  
  Image(..)  
  Column {  
    Text(..)  
    Text(..)  
  }  
}
```



UI-treet (forts.)



Eksempel: En enkel teller

```
@Composable
```

```
fun Counter(modifier: Modifier = Modifier) {
```

```
    Column(modifier = modifier, horizontalAlignment = Alignment.CenterHorizontally) {
```

```
        var count by remember { mutableIntStateOf( value: 0) }
```

```
        Text(text = "$count", style = MaterialTheme.typography.displayLarge)
```

```
        Row(
```

```
            horizontalArrangement = Arrangement.Center,
```

```
            verticalAlignment = Alignment.CenterVertically
```

```
        ) {
```

```
            listOf(1,5,10).forEach {
```

```
                Button(modifier = Modifier.padding(2.dp), onClick = { count += it }) {
```

```
                    Text(text = "+$it")
```

```
                }
```

```
            }
```

```
        }
```

```
    }
```

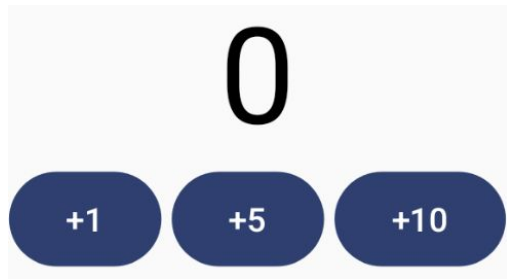
```
}
```

Teller-komponenten - oppbygging

```
@Composable
fun Counter(modifier: Modifier = Modifier) {
    Column(modifier = modifier, horizontalAlignment = Alignment.CenterHorizontally) {

        var count by remember { mutableIntStateOf( value: 0) }

        Text(text = "$count", style = MaterialTheme.typography.displayLarge)
        Row(
            horizontalArrangement = Arrangement.Center,
            verticalAlignment = Alignment.CenterVertically
        ) {
            listOf(1,5,10).forEach {
                Button(modifier = Modifier.padding(2.dp), onClick = { count += it }) {
                    Text(text = "+$it")
                }
            }
        }
    }
}
```



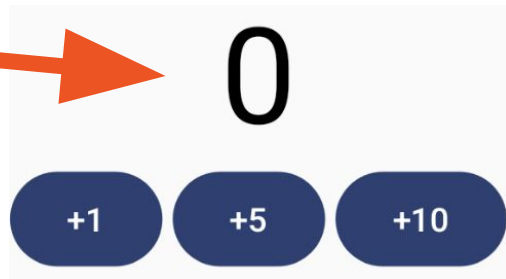
Teller-komponenten - oppbygging

```
@Composable
fun Counter(modifier: Modifier = Modifier) {
    Column(modifier = modifier, horizontalAlignment = Alignment.CenterHorizontally) {

        var count by remember { mutableIntStateOf( value: 0) }

        Text(text = "$count", style = MaterialTheme.typography.displayLarge)

        Row(
            horizontalArrangement = Arrangement.Center,
            verticalAlignment = Alignment.CenterVertically
        ) {
            listOf(1,5,10).forEach {
                Button(modifier = Modifier.padding(2.dp), onClick = { count += it }) {
                    Text(text = "+$it")
                }
            }
        }
    }
}
```



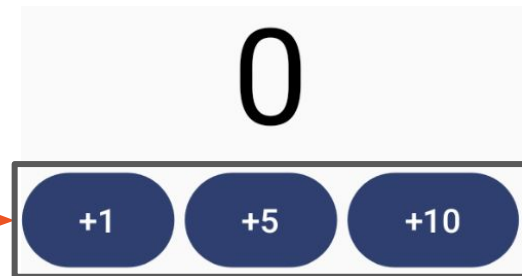
Teller-komponenten - oppbygging

```
@Composable
fun Counter(modifier: Modifier = Modifier) {
    Column(modifier = modifier, horizontalAlignment = Alignment.CenterHorizontally) {

        var count by remember { mutableIntStateOf( value: 0) }

        Text(text = "$count", style = MaterialTheme.typography.displayLarge)

        Row(
            horizontalArrangement = Arrangement.Center,
            verticalAlignment = Alignment.CenterVertically
        ) {
            listOf(1,5,10).forEach {
                Button(modifier = Modifier.padding(2.dp), onClick = { count += it }) {
                    Text(text = "+$it")
                }
            }
        }
    }
}
```



Teller-komponenten - oppbygging

```
@Composable
fun Counter(modifier: Modifier = Modifier) {
    Column(modifier = modifier, horizontalAlignment = Alignment.CenterHorizontally) {

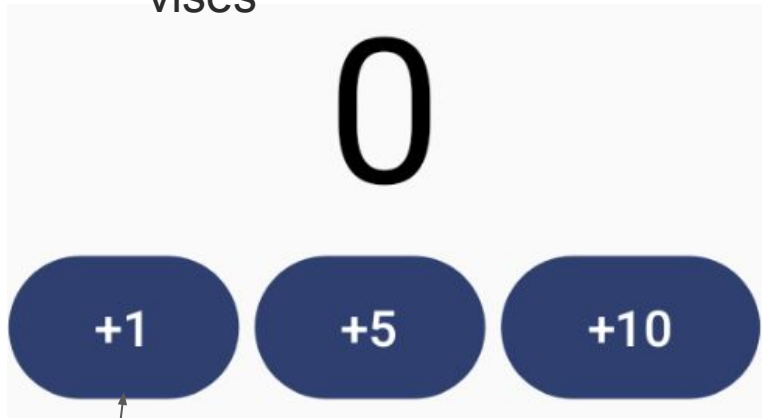
        var count by remember { mutableIntStateOf( value: 0) }

        Text(text = "$count", style = MaterialTheme.typography.displayLarge)
        Row(
            horizontalArrangement = Arrangement.Center,
            verticalAlignment = Alignment.CenterVertically
        ) {
            listOf(1,5,10).forEach {
                Button(modifier = Modifier.padding(2.dp), onClick = { count += it }) {
                    Text(text = "+$it")
                }
            }
        }
    }
}
```

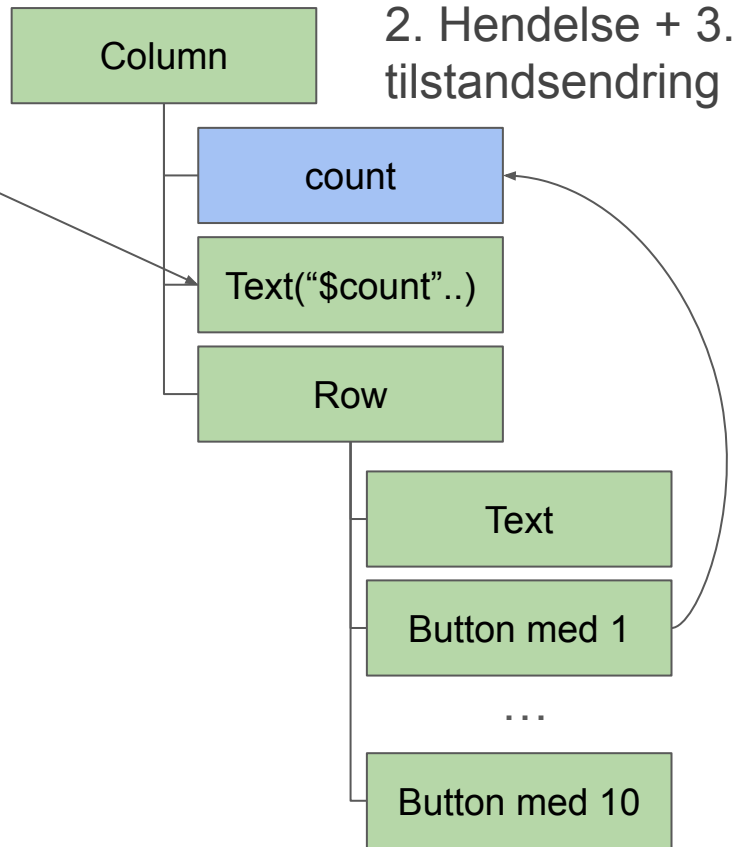


Teller-komponenten - hva skjer?

4. Rekomposisjon + data flyter ned til komponent og ny tilstand vises



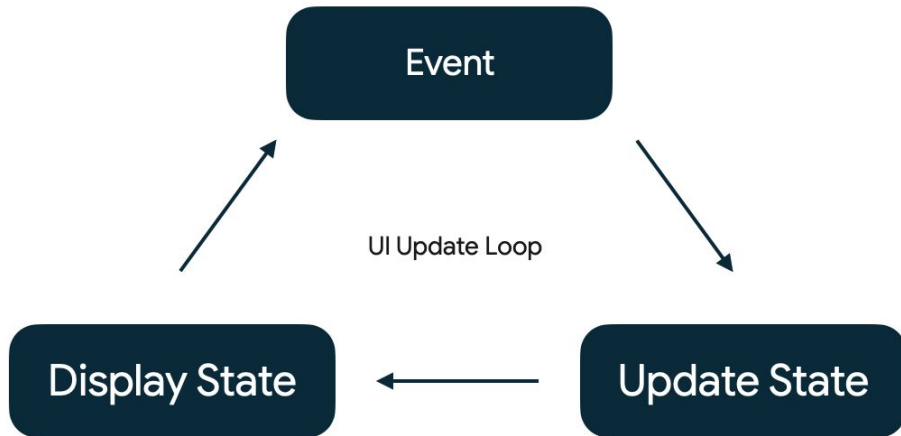
1. Trykk på knapp



UI update loop

State definerer hva som vises i brukergrensesnittet til enhver tid

- Tilstand *er*
- Hendelser *skjer*



Tilstand

- En app sin “tilstand” er enhver verdi som kan endres over tid. Dette kan være alt fra en database til en variabel i en klasse
- Noen eksempler er:
 - Hva telleren er på nå
 - Hvor langt det er scrollet i en liste
 - Meldinger en har mottatt
 - Om noe er “expanded” eller ikke
- Compose har et eget system for å håndtere endringer i tilstand

Tilstand (forts.)

- Observerbar datatype
- Endringer i en State sin value vil forårsake rekomposisjon i Composables som bruker denne tilstanden
- `mutableStateOf` er en måte å opprette et observerbart objekt av *MutableState<T>*-typen

```
interface MutableState<T> : State<T> {  
    override var value: T  
}
```

remember

```
var value by remember { mutableStateOf(default) }
```

- *remember* lagrer ting i komposisjonen
- Verdiene inni remember forblir gjennom en rekomposisjon
- Den vil forsvinne når komponenten den er inneholdt i blir borte fra komposisjonen
- Du kan benytte deg av en “remember”-verdi som parameter til andre Composables eller i logikk for å endre hvilke composables vises
- Ofte brukes *remember* for å ta vare på en MutableState

State hoisting

- Gjøre en Composable tilstandsløs (stateless)
- Skille visning av tilstand fra produksjon av tilstand
- Et mønster der en flytter tilstand ut av en Composable “opp” til den som kaller på en Composable
- Normalt sett bytter vi ut tilstandsvariabelen med to parametere
 - **value: T**: den nåverende verdien som skal vises
 - **onValueChange: (T) -> Unit**: forespør at verdien skal endres, der T er den foreslåtte nye verdien

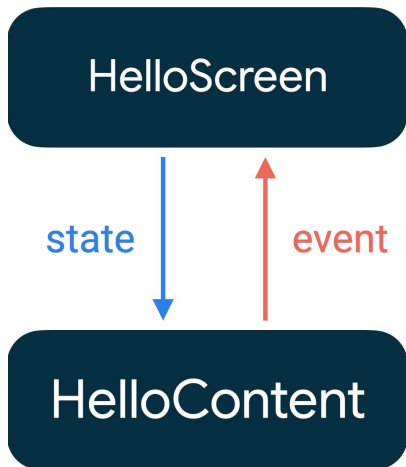
State hoisting (forts.)

```
@Composable
fun HelloScreen() {
    var name by rememberSaveable { mutableStateOf("") }

    HelloContent(name = name, onNameChange = { name = it })
}

@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello, $name",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium
        )
        OutlinedTextField(value = name, onValueChange = onNameChange, label = { Text("Name") })
    }
}
```

StateOverviewSnippets.kt



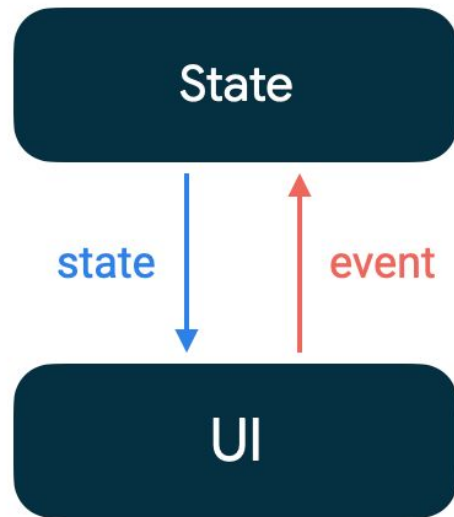
Egenskaper ved “hoisted state”

Noen egenskaper ved “hoisted state”:

- **Single source of truth**
- **Encapsulated**, kun Composables med tilstand kan endre sin tilstand
- **Shareable**, tillater at en kan lese tilstanden i en annen Composable
- **Interceptable**, kan velge å ignorere eller håndtere hendelser før tilstanden endres
- **Decoupled**, tilstanden er frakoblet brukergrensesnittet

Unidirectional data flow (UDF)

- Design pattern hvor tilstand flyter ned og hendelser flyter opp
- Vanlig flyt:
 - Event sendes oppover
 - EventHandler vil kanskje endre tilstanden
 - Den som holder på tilstanden vil sende det “nedover” og brukergrensesnittet vil vise det



Unidirectional data flow - Fordeler

- Kan separere visning av tilstand i brukergrensesnittet fra delene av appen som lagrer og endrer tilstand
- Noen andre fordeler:
 - **Testbarhet**
 - **Innkapsling av tilstand:** det finnes kun én sannhet for tilstanden til Composablen
 - **Konsistent brukergrensesnitt:** Alle tilstandsendringer vil umiddelbart bli reflektert i brukergrensesnittet ved å bruke observerbare state holders

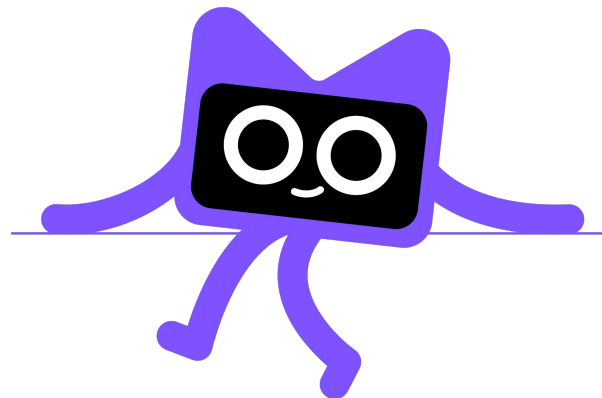
Navigasjon i Android og Compose

- Navigasjon i Android består av flere komponenter
 - **NavController**, som er stateful og holder styr på “backstack-en” av composables som er skjermene i appen og tilstanden til hver skjerm.
 - **NavHost**en kobler NavControlleren til en graf som spesifiserer composable-destinasjonene en skal kunne navigere til. Innholdet i NavHost-en blir automatisk rekonponert når en navigerer mellom Composables.
 - Hver Composable-destinasjon i navigasjonsgrafen er assosiert med en **rute**.
- <https://developer.android.com/jetpack/compose/navigation>

Oppsummering

Hva har vi gått gjennom til nå? - Kotlin

- Variabler
- Mutable / Immutable
- String templates
- Interoperabilitet med Java
- Funksjoner
- Kontrollflyt
- Null safety
- Klasser og objekter
- Samtidighet og parallellitet (Coroutines)



Hva har vi gått gjennom til nå? - Android

- Introduksjon til Android
- Intro til Jetpack Compose
- Composable-funksjoner og noen eksempler på dette
- Previews
- Compose-state og *remember*
- Unidirectional Data Flow
- Gradle
- Navigasjon



Videre i IN2000

mer **Android!**

- Applikasjonsarkitektur
 - Design patterns
 - Live-koding

mer **Kotlin!**

- Funksjonell programmering
 - Høyere-ordens funksjoner og lambdaer
 - Operasjoner på Collections
- Gjesteforelesning fra JetBrains