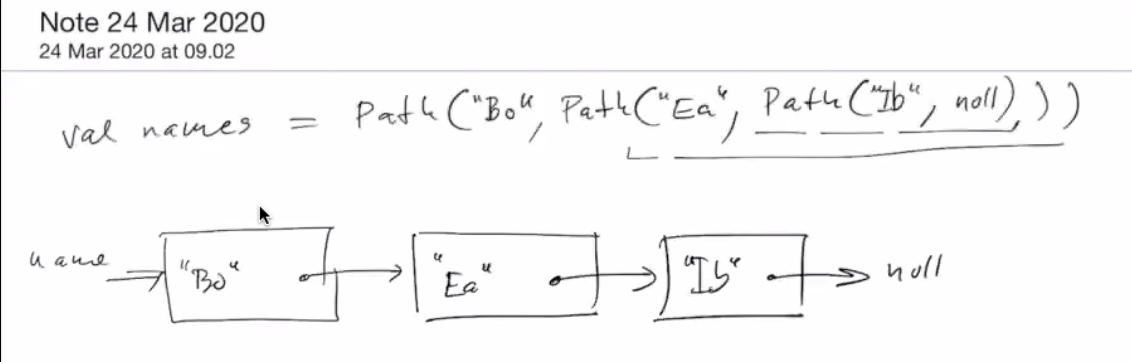
**Coroutines, god kodeskik og forskelle mellem Kotlin og Java:**

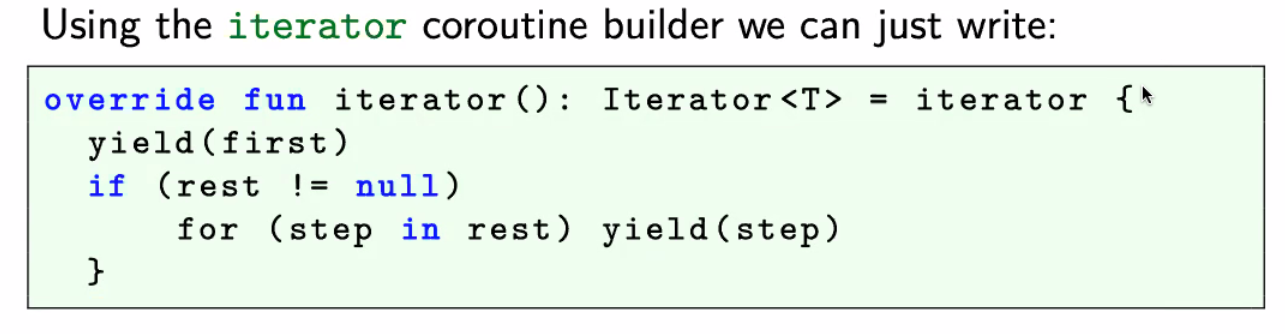
Kotlin har en anden måde at tilgå properties.  
Opløser konflikter mellem struktur og ansvar.  
Et letvægtsalternativ til tråde  
  
To måder at tilgå coroutiner  
Inside out – i stedet for at skulle afvikle koden indefra kan vi styre den udefra.  
Concurrency - …  
  
LISP et af de ældste højniveau-programmeringssprog  
  
Kotlin - Hvis man putter val eller var ind i konstruktøren, bliver de automatisk til properties  
  
For loop med en LISP lignende linked list:  


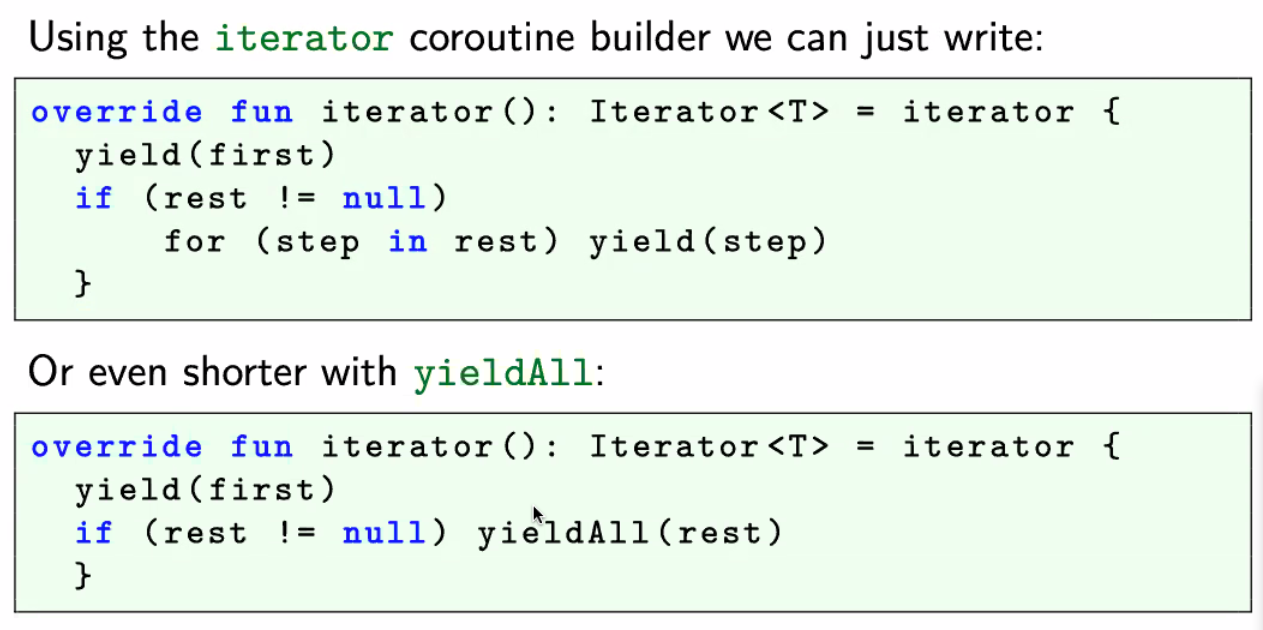
Typisk laver man integrators som inner class for classes der implementerer itterable.  
- Itterable skal have en metode der heder iterator

- Iterator skal have to metoder der hedder next og hasnext

- Has next her rimelig nem at implementere, da vi bare skal returnere en boolean for om rest er forskellig fra null.

- Next er lidt mere kompliceret, da vi er nødt til at gemme hvad end der er den færreste neste i rest i en variabel også selv om det er null, hvorved vi vil forces til at få en exception. Herefter returnerer vi rest, hvis vi har noget at returnere.

Hvis vi kunne skrive en iteration inde fra path classen, i stedet for at lave en ful LIST lignende class implantation af itterable, vil det gøre vores forrige arbejde meget nemmere. Her kan vi returnerer first fra rest, hvis forskellig fra null, indtil der ikke er flere elementer.  


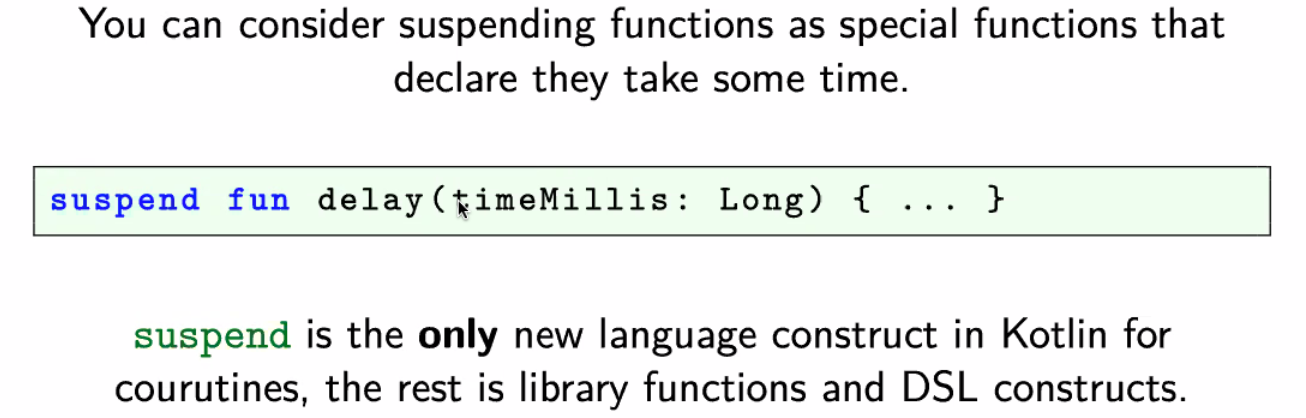
Hvis vi implementerer vores itteration med coroutiner, holder coroutinerne styr på vores state inde i iterationen.  
  
Yield overgiver os ’magten’ over det valgte element, i dette tilfælde first både i starten, og i den recursive iteration hvis rest er forskellig fra null. Dette er et godt eksempel på hvordan vi kan bygge en iterator indefra og gøre den meget kortere end en class implantation af iterator  


Coroutiner er for det meste implementeret i kode via DSL-strukturer, men der er brug for enkelte byggeblokke.

CoroutineContext er hvor Kotlin placere information om hvor coroutinen var efterladt sidste gang og hvad state er for de lokale variabler, en slags switch der siger hvor langt vi nåede og gemmer de lokale variabler og den daværende tilstand, fryder tilstanden, hvor nåede jeg til. Denne context er vigtig.

Suspended som keyword kan bruges til en efterfølgende eller parallel eksekvering. Bruges typisk når vi har at gære med noget der kan tage lang tid, som f.eks. web requests. Typisk har suspended som keyword en CoroutineContext med som parameter.

CoroutineScope som keyword bliver brugt til at holde styr på coroutiner og afventer at de bliver færdiggjort eller bliver afbrudt. Man kan godt lave sin egne coroutine scopes der kan indkapsulerer coroutiner.

Suspending funktionen kan suspendere eksekveringen af den nuværende coroutine uden at blokere den nuværende tråd.  


Inde i en suspended funktion har vi styr på hvilken rækkefælge der bliver eksekveret i det udfald.

Inde fra en suspending funktion, kan vi lave sekventielle funktionskald

En funktion bliver ikke asyncron ved brug af suspending keyworded alene, men kun hvis de specifikt bruges sådan inde fra funktionen.

**Suspending funktioner**

En coroutine builder er en simple funktion der kan bygge en ny coroutine som kan køre suspending funktioner. De kan blive kaldt fra normale non suspending funktioner fordi de ikke automatisk selv er suspending. På denne måde virker de som en bro til suspending funktionskald. Den kan dog kun blivekaldt inde i et coroutineScope. Her kan man lave en bridge mellem noget der er suspendable og noget der ikke er suspendable.

Man kan kun lave susped på et scope der har en coroutine at arbejde på.

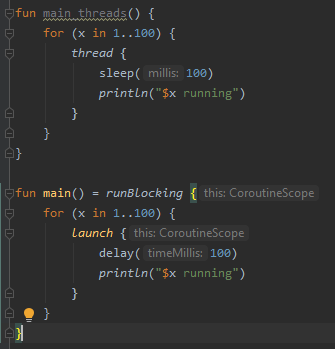
runBlocking – venter på at alle coroutiner færdiggøres før vi aflevere ’magten’ tilbage

launch – vi kan køre noget uden at afvente noget der returneres.

async – vi køre noget og afventer at vi får noget tilbage.

Await – afventer til det overstående er færdig med at eksekvere.

Det minder meget om tråde, på den måde som GlobalScope giver magen videre på, hvis vi bliver forsinkede i eksekveringen mellem funktioner.

Coroutine oprettelsen og eksekveringen af coroutiner er så hurtig at de hvis kært i et for loop vil blive returneret i rækkefølge uden at være synkroniseret. Dette ligger i modsætning til tråde, der hvis eksekveret i et for loop returneres i en blandet orden.  


Der er rigtig stor forskel på hvor hurtigt det tager at eksekvere en række tråde i forhold til en række coroutiner. Det gain man får ved at lave coroutiner betaler sig virkelig når man har så mange processer at det bliver dybere at oprette tråde end det gør at lave coroutiner.

Hvis noget har et CoroutineScope, har vi at gøre med en coroutine.