# Teknologi

## Trådprogrammering (Karsten)

En proces afvikles af operativsystemet (eller den virtuelle maskine; Java Virtual Machine i vores tilfælde) i sin egen *execution context* der indeholder processens tilstand under afvikling. Den enkelte proces kan udnytte moderne processorers muligheder for parallelisme (concurrency) ved at uddelegere arbejdsopgaver til tråde—”miniprocesser”—som alle deler adgang til processens execution context.

### Udfordringer i concurrency

Fordi trådene deler samme tilstand og afvikles parallelt, medfører brugen af concurrency nogle problemer: Hvis flere tråde arbejder på samme del af tilstanden, kan vi opleve *thread interference*; at deres operationer interfererer og overskriver hinanden med uventede resultater til følge.

Et andet problem er *memory inconsistency* hvor flere tråde ser forskellig data som burde være ens i den samme tilstand. Til håndtering af dette arbejder vi med *happens-before-*garantier. Når en happens-before-garanti er givet, er vi forsikret om at ændringer i tilstanden udført af én operation er synlige for en efterfølgende operation. Det er fx tilfældet når vi kalder Thread.start—effekten af alle operationer inden kaldet er synlige for alle operationer i den nye tråd der startes. [JT]

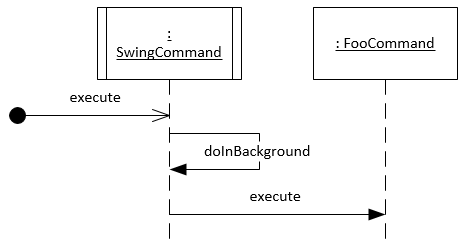
### Synkronisering

For at undgå thread interference og give happens-before-garantier kan vi indføre synkronisering på metoder og statements. Synkronisering medfører at en intrinsic lock—intern lås—sikrer at kun én tråd kan afvikle en given metode eller statement ad gangen.

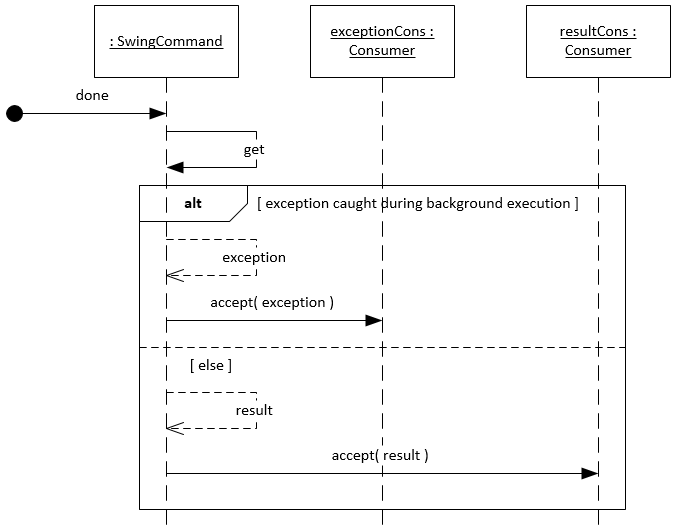
Synkronisering er en løsning på de første problemer; desværre indfører brugen af locks nye problemer vi må være opmærksomme på. En af de mest kendte er deadlock: en tilstand vi kan risikere hvor afviklingen af to eller flere tråde går i stå fordi de alle venter på hinanden. Omvendt beskriver den mere sjældne livelock en situation hvor trådene ikke går i stå, men konstant giver plads til hinanden uden at nogen af dem får udført deres opgave. Et andet fænomen er starvation hvor vi risikerer at en tråd må vente meget længe på adgang til en resurse fordi den optages af en eller flere andre ”grådige” tråde. [JT]

### SwingWorker

Da vi i dette projekt arbejder med en Swing GUI, har vi valgt at gøre brug af javax.swing.SwingWorker til håndtering af vores tråde. [JT2] Det særlige ved SwingWorker er dens doInBackground- og done-metoder. Når vi kalder execute på en SwingWorker, oprettes automatisk en ny tråd (SwingWorker gør brug af Executors internt). I den nye tråd kaldes doInBackground som udfører vores tidskrævende operation. Når doInBackground har returneret, kaldes done automatisk i EDT (Event Dispatch Thread); tråden hvor alle Swing events behandles. Vi kan således komme helt uden om manuel synkronisering, blot vi holder vores doInBackground-operationer indkapslet som isolerede operationer uden brug af delt tilstand. SD-diagrammet i figur NNN viser hvordan vi i EDT starter udførelse af en kommando (a) og håndterer resultatet fra den efterfølgende (b).



(a)



(b)

*Figur NNN: Udførelse af FooCommand med SwingCommand og Consumer*

Er det en event i brugerfladen der starter udførelse af FooCommand, instantierer brugerfladen to Consumer-objekter (functional interface); den ene til behandling af det forventede resultat, den anden til behandling af en evt. exception. Brugerfladen sender disse consumers med kommandokaldet til controller (via facade), og controller opretter en instans af SwingCommand (vores nedarvning af SwingWorker) der modtager consumer-instanserne. Således kan SwingCommand—ved kald til done—undersøge om et resultat kom tilbage fra FooCommand.execute, eller der blev fanget en exception under udførelsen, og kalde accept på den tilsvarende consumer.