Objektorientert programmering av datastrukturer og grensesnitt (Java)

Knut W. Hansson

SKRIFT-SERIEN
Nr. 7

2014

Objektorientert programmering av datastrukturer og grensesnitt (Java)

Knut W. Hansson

Skriftserien fra Høgskolen i Buskerud og Vestfold nr 7/2014

Om publikasjonen:
Ved Høgskolen i Buskerud og Vestfold, bachelorstudiene i IT, undervises kurset "Objektorientert programmering av datastrukturer og grensesnitt ("Java 2")" som gir 7,5 studiepoeng. Studentene har da tidligere hatt ganske mye objektorientert programmering, delvis med Visual Basic (16,5 stp) og delvis med Java (7,5 stp). De har også hatt et kurs i objektorientert analyse og design (UML – 7,5 stp) og databaser (7,5 stp). Kurset "Java 2" er således et kurs for viderekommende og forutsetter betydelige kunnskaper innen objektorientering og Java-syntaks. Publikasjonen inneholder forfatterens forelesninger fra det nevnte kurset, samt noe tilleggsstoff for spesielt interesserte. Det er laget en egen elektronisk lærerressurs med løsningsforslag til mange av oppgavene.

Om forfatteren:
Knut W. Hansson er førstelektor ved Høgskolen i Buskerud og Vestfold, Campus Ringerike, Handelshøgskolen og fakultet for samfunnsvitenskap.

Objektorientert programmering av datastrukturer og grensesnitt (Java)
Knut W. Hansson

© Høgskolen i Buskerud og Vestfold / Knut W. Hansson, 2014

Skriftserien fra Høgskolen i Buskerud og Vestfold nr 7/2014

Skriftserien kan lastes ned fra http://bibliotek.hbv.no/skriftserien

ISSN: 1894-7522 (online)
ISBN: 978-82-8261-027-8 (online)

Omslag: Kommunikasjonsseksjonen, HBV

Utgivelser i HBVs skriftserie kan kopieres fritt og videreformidles til andre interesserte uten avgift. Navn på utgiver og forfatter(e) angis korrekt. Det må ikke foretas endringer i verket.

Sammendrag

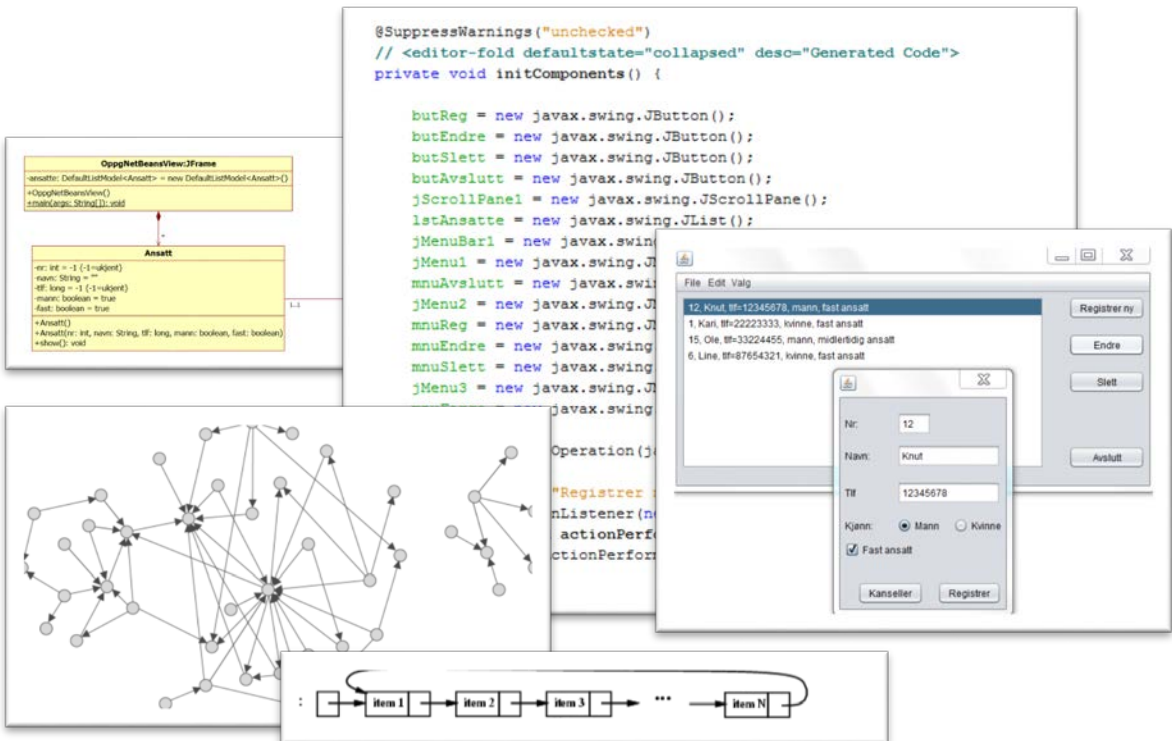
Ved Høgskolen i Buskerud og Vestfold, bachelorstudiene i IT, undervises kurset "Objektorientert programmering av datastrukturer og grensesnitt ("Java 2")" som gir 7,5 studiepoeng. Studentene har da tidligere hatt ganske mye objektorientert programmering, delvis med Visual Basic (16,5 stp) og delvis med Java (7,5 stp). De har også hatt et kurs i objektorientert analyse og design (UML – 7,5 stp) og databaser (7,5 stp).

Kurset "Java 2" er således et kurs for viderekommende og forutsetter betydelige kunnskaper innen objektorientering og Java-syntaks. Det skal bringe studentene videre i objektorientert programmering med Java og mer avanserte mekanismer med Java som eksempel. Noe av det som omtales er spesielt for Java-språket, men mye finnes tilsvarende i andre objektorienterte programmeringsspråk. Det bør altså ha stor overføringsverdi til disse.

Kurset anvender IDE-verktøyet NetBeans. Tidligere har studentene brukt Visual Studio og Eclipse. NetBeans er altså det tredje IDE de bruker. Også her vil det være overføringsverdi til andre IDE-verktøy.

Denne boken inneholder alle forelesningene mine fra det nevnte kurset, samt noe tilleggsstoff for spesielt interesserte.

Det er laget en egen elektronisk lærerressurs med løsningsforslag til mange av oppgavene.



Forord

Introduksjon til boken

Denne boken inneholder alle mine forelesninger om objektorientert programmering av datastrukturer og grensesnitt ved Høgskolen i Buskerud og Vestfold. Det omfatter også mange andre mekanismer i Java. Det skal ikke være nødvendig med ytterligere litteratur for å følge dette kurset. Boken kan også leses på egen hånd.

Boken inneholder også en del tilleggsstoff som ikke direkte er pensum i det nevnte kurset, men som danner interessant bakgrunn og/eller kan være av spesiell interesse. Det er spesielt merket. Oppgavene kan løses uten å tilegne seg disse delene av stoffet.

Uansett om du følger det nevnte kurset eller leser boken på egen hånd, bør du gjøre *alle* oppgavene (det er én til hvert kapittel). Løs også caset til slutt i boken. Det er ved å *gjøre* det du har lest om, at du lærer best.

Forkunnskaper

Boken forutsetter betydelige kunnskaper om objektorientert programmering generelt og en god del kunnskaper i Java spesielt.

Nytteverdi

Java er et av de mest brukte programmeringsspråk for tiden[1] og objektorientert programmering generelt er svært utbredt. I en jobbsituasjon er det derfor stor sannsynlighet for at kunnskapene vil komme til nytte.

Det vil alltid være svært mye innen Java som studentene ikke får undervisning i, men sammen med all annen undervisning de får i objektorientert og imperativ programmering bør de – etter mitt syn – kunne gjøre god nytte for seg med en gang og ha et svært godt grunnlag for å lære videre selv i en arbeidssituasjon.

---

[1] De som tør å mene noe om populære programmeringsspråk kommer til forskjellige resultater, men det er noen programmeringsspråk som går igjen høyt opp på listene:

a. Java
b. PHP
c. C
d. C++
e. C#
f. Objective-C
g. Som mer usikre "runners up" vil jeg trekke frem Perl, Ruby, Python og Visual Basic.

Man kan merke seg at de aller fleste er objektorienterte. Se f.eks. http://redmonk.com/sogrady/2013/02/28/language-rankings-1-13/

Innhold

Om motivasjon 1
Kapittel 1 – GUI i Java 3
GUI: AWT eller Swing? 3
Forholdet mellom JVM og OS 4
Lage grafisk brukergrensesnitt 5
Ekstra: Prinsipper for JFrame 6
Bestemme LayoutManager og andre egenskaper ved vår JFrame 6
Legge til komponenter 7
Lyttere 8
Oppsummering 9
Ekstra: Swing "L&F" Bilder 11
Ekstra: Oversikt over Swing 12
Oppgave til kapittel 1 16
Kapittel 2 – NetBeans 18
Noen demovideoer om NetBeans 18
Installere og tilpasse NetBeans 18
Tips ved bruk av NetBeans 19
Bruke NetBeans 21
Ekstra: EasyPMD for NetBeans 26
Ekstra: Bruk av komponentbiblioteker i NetBeans 27
Ekstra: Tilpasse Windows Explorer til NetBeans 28
Oppgave til kapittel 2 28
Kapittel 3 – JavaDoc 32
Bruk av Java API Dokumentasjon 32
Lage egen JavaDoc 33
JavaDoc tagger 34
Generere din JavaDoc 35
Eksempel 35
Ekstra: Likhet og tilordning i Java, argumenter/parametre 38
Oppgave til kapittel 3 42
Kapittel 4 – Abstrakte datatyper (ADT) 44
ADT figurer 46
Forskjellen på lister og arrays 47
Oversikt over Java-grensesnitt for samlinger 49
Oversikt over Java-klasser for samlinger 49
Oppgave til kapittel 4 50
Kapittel 5 – Lister inkl. Vector i Java 52
Oppgave til kapittel 5 54
Kapittel 6 – Maps, set og deque 56
Maps 56
Set 57
Deque 58
Ekstra: Performance for de forskjellige samlingstypene 58
Ekstra: Litt enkel kode for maps 59
Oppgave til kapittel 6 60
Kapittel 7 – Rekursjon 62
Ekstra: Litt repetisjon om funksjoner (for sikkerhets skyld) 62
Rekursive funksjoner 62
Demo 63
Eksempel: Potens rekursivt 63
Eksempel: Finn største i en liste 63
Eksempel: Multiplikasjon 64
Eksempel: Omgjøring fra desimaltall til binært tall 64
Eksempel: "Traversere" trær 65

Eksempel: Shift-operasjon ... 66
Oppgave til kapittel 7 ... 67
Kapittel 8 – Databaser ... 69
ODBC eller JDBC? ... 69
1. MySQL ... 70
2. Oracle database ... 73
3. Java DB database ("derby") ... 75
Sql-setninger for Statement (uansett database) ... 76
Andre databaser ... 77
Ekstra: Kort oversikt over gangen i koden ... 77
Ekstra: Bruke NetBeans Database Explorer ... 77
Ekstra: Objektorientert database ... 80
Oppgave til kapittel 8 ... 81
Kapittel 9 – Klient-tjener ... 83
Sockets ... 83
Byte-strømmer mellom sockets ... 84
Skrive til en byte-strøm ... 84
Lese fra en byte-strøm ... 84
Kombinasjonen lese/skrive ... 84
Eksempel ... 85
Ekstra: Serialisering av objekter ... 86
Oppgave til kapittel 9 ... 87
Kapittel 10 – Testing ... 89
Generelt om testing ... 89
Typer av tester ... 89
Regresjonstesting ... 90
Om TTD ... 90
Om JUnit ... 90
Slik gjør vi ... 91
Forenkling av testkoden ... 96
Ekstra: Teste private metoder ... 100
Ekstra: Unit Testing Checklist ... 101
Oppgave til kapittel 10 ... 102
Kapittel 11 – Java-bibliotek ... 105
Hva er et bibliotek ... 105
Lage biblioteket ... 105
Bruke biblioteket ... 105
Demo: Klasserom ... 106
Oppgave til kapittel 11 ... 106
Kapittel 12 – Tråder i Java ... 108
Hva er tråder (threads)? ... 108
En eller flere tråder i et program ... 108
Metode 1: Skape flere tråder tradisjonelt med *Runnable* eller *Thread* ... 109
Synkronisering ... 110
Oppdatering av GUI med bruk av tråder ... 110
Metode 2: SwingWorker (som ofte er greiere å bruke enn *Runnable* og *Thread*) ... 111
Et litt større eksempel ... 112
Oppgave til kapittel 12 ... 115
Kapittel 13 – Noen nyttige språkelementer ... 117
Streams ... 117
Lambda expressions ... 118
Klasse-metoder (static) i grensesnitt ... 120
Default-metoder i grensesnitt ... 121
Fork/join ... 122
Litt om metoder og try-catch ... 124
Metoder som kan returnere uten resultat ... 127

Ekstra: Optional som parameter 128
Oppgave til kapittel 13 129
Kapittel 14 – Ekstra fagstoff for egen lesning 130
Synlighet i Java V14 130
Java Regular Expressions (regex) 130
Om StringBuilder og StringBuffer 131
Litt kodeteori fra Internett 132
Om kryptisk kode 139
Pair Programming 139
Exceptions – en diskusjon 140
Annen bruk av try-catch 144
Utfordring 14A til kapittel 14: Er listen rekursiv? 144
Utfordring 14 B til kapittel 14: Overlapper to rektangler? 146
Avsluttende case – heldags prosjektoppgave 147

Om motivasjon

Det kan være vanskelig å finne motivasjon til å lære seg noe vanskelig, f.eks. programmering.

Kanskje motiverer du deg ved å tenke at du må lære dette for å få en bra karakter eller for å unngå å stryke. Dette kalles *"ytre motivasjon"*. Amerikanere liker å snakke om det som "carrot and stick" motivasjon. "Gjør du dette så skal du få..." og "gjør du ikke dette så...".

Dessverre viser forskning at dette bare virker når arbeidsoppgaven er enkel og rutinemessig. Hvis oppgaven er kompleks og innebærer kreativitet og problemløsning så virker slik "carrot & stick" motivasjon mot sin hensikt.

Hva er det da som virker for slike komplekse oppgaver som f.eks. å lære seg programmering? Det viser seg å være tre prinsipper som alle kalles *"indre motivasjon"*:

Autonomi – friheten til selv å velge hvordan og når
Mestring – følelsen av å få det til
Mening – følelsen av at dette er meningsfullt, har en hensikt

Det har lenge vært kjent at *indre* motivasjon er mye mer effektivt enn *ytre* motivasjon. Folk kan til dels fristes og tvinges til å gjøre noe, men det blir mye bedre og løses raskere hvis personen faktisk *ønsker* å gjøre det av egne grunner. Hvorfor lærer f.eks. noen seg å spille gitar på fritiden? Hvorfor trener de ballspill? Ytterst få for betalt for treningen (selv om de kanskje får betalt når de først har lært seg det) men de gjør det allikevel. Jeg er sikker på at også du har slike aktiviteter som du gjør jevnlig helt uten "carrot and stick" motivasjon.

Spørsmålet blir jo da hvordan en student kan få til dette? Det kan du sikkert finne dine egne svar på, men jeg har noen forslag:

Autonomi: Det er opp til deg når du vil lære et punkt i pensum, riktignok innenfor visse frister (for levering, innen eksamen osv.). Det er også opp til deg hvordan du vil lære deg det – lese, øve, diskutere eller annet.

Konklusjon: ***Autonomi* har du allerede i stor grad, men du må utnytte den.**

Mestring. Hvis du faktisk jobber med emnet, vil du også lære deg å mestre det. Trikset er da å legge bevisst merke til mestringen. Hvis du ikke legger merke til mestringen får den heller ingen positiv effekt. Hver ukeslutt kan du f.eks. tenke igjennom (reflektere over) hva du faktisk har lært denne uken. Hvis du virkelig vil understreke det, kan du skrive det opp. Hva med å legge ut listen i Facebook? Eller henge den opp hjemme?

Konklusjon: ***Mestring* er lett for deg å oppnå, men du må legge merke til det.**

Mening. Tenk igjennom hva dette skal være godt for. Dette er ikke noe du skal lære fordi faglæreren din krever det. Jo, det kan jo være at han krever det, men *hvorfor* krever han det? Han har tenkt igjennom hva studenter bør lære og kan begrunne det. Viktigere er det for deg å finne meningen *for deg*. Det kan vel fort bli bruk for det i en fremtidig jobb, kanskje?

Konklusjon: ***Mening* er en personlig sak, den må du selv finne ut av.**

Jeg kan anbefale å bruke noen minutter på en animasjonsvideo som forklarer prinsippene svært tydelig, men husk å tenke igjennom hvordan dette kan være relevant for *deg*: http://www.thersa.org/events/rsaanimate/animate/rsa-animate-drive

Kapittel 1 – GUI i Java

GUI: AWT eller Swing?

Ettersom Java kjører i en virtuell maskin må Sun/Oracle velge om de vil bruke det lokale operativsystemets API til grafikken eller tegne skjermen selv. De har laget to biblioteker og lar oss velge – men hva bør vi velge? Nedenfor har jeg redegjort for valgene og noen konsekvenser.

AWT (Abstract Window Toolkit)

Biblioteket AWT er det opprinnelige (fra JDK 1.0). Det benytter APIer i det lokale OS til å skape grafikken, plassere vinduer foran hverandre osv. APIene som en metode bruker, kalles metodens "peer" (kollega). "Peers" lastes inn og øker størrelsen på det kjørende programmet. AWT sies derfor å være "heavyweight".

Når man skal lage et slikt bibliotek som AWT, er man avhengig av å velge muligheter som finnes i (nesten) alle OS. Det gir restriksjoner som Sun/Oracle må forholde seg til. Allikevel meldes det at enkelte ting ikke virker på visse OS.

Alle nettlesere har implementert AWT i sin håndtering av Applets.

Swing

Biblioteket Swing er kommet til bare litt senere (JDK 1.2). Det er skrevet mest mulig i Java. Også dette biblioteket er avhengig av APIer i OS, men det bruker bare mer grunnleggende APIer som ren grafikkhåndtering og ikke mer overordnede som håndtering av klikk, plassering av vinduer o.l. Det ordner Swing selv. Siden det da ikke er nødvendig å laste "peers", kalles Swing "lightweight".

Siden Swing gjør mer selv, får den ikke utnyttet hardware like effektivt, og den er derfor kjent som tregere enn AWT og trenger kraftigere maskiner.

Swing vil som default se likt ut uansett hvilket OS det kjører på (*CrossPlatformLookAndFeel* kalt "Java L&F" eller "Metal"). Imidlertid kan det også settes til å "etterlikne" det lokale OS (*SystemLookAndFeel)* som bestemmes ved run time. Swing bare *emulerer* kontrollene som finnes lokalt og derfor virker ikke alltid *SystemLookAndFeel* helt. F.eks. vil Apple OS alltid vise sin egen "L&F" uansett, men den kan ikke lovlig brukes på andre maskiner.

Siden Swing er "lightweight", må det vedlikeholdes når et OS endrer "L&F". Det kan føre til at et kompilert program ikke "følger med" i "L&F"-utviklingen. AWT som er "heavyweight", vil automatisk følge med. På den annen side får AWT problemer hvis grensesnittet på APIer endres (men det vil naturligvis OS-leverandøren forsøke å unngå).

Swing har alle komponentene som AWT har, og i tillegg flere avanserte som "tree view", "list box", "tabbed pane", "tool tips", ikoner og andre. Det hevdes at Sun/Oracle arbeider mest med Swing for tiden men en av grunnene til det kan være at de jobber for å få opp hastigheten.

Mange nettlesere kan ikke vise applets med Swing uten å ha installert Java plugin. Det gjelder i alle fall Firefox og MSIE. For brukere kan det være et problem.

AWT eller Swing – valgets kval

"Alle" er enige om at man ikke bør blande AWT og Swing i samme applikasjon/applet. De tegner grafikken på forskjellig måte rent prinsipielt og vil fort ødelegge for hverandre.

Noen momenter i valget:
- ✓ *Fart:* AWT utnytter operativsystemets hastighet og er raskere enn Swing.
- ✓ *Applets:* Nettlesere viser applets med AWT uten problemer, applets med Swing vil kreve at nettleseren har Java plugin (og det må brukeren installere).
- ✓ *Utseende/oppførsel ("L&F"):* AWT benytter lokale "peers" og vil derfor alltid likne på andre applikasjoner lokalt. Swing kan settes til å etterlikne lokal "L&F" mer eller mindre vellykket, men kan også bruke Javas egen "L&F". Altså: Swing har mer fleksibelt "L&F"[2] (se senere i kapittelet, side 11).
- ✓ *Komponenter:* Siden AWT må velge et "minste felles multiplum" fra mange OS, får det færre muligheter enn Swing som lager det meste selv. Swing har følgelig flere komponenter.
- ✓ *Andre biblioteker:* Flere andre biblioteker som f.eks. Borland benytter Swing. Hvis du skal bruke dem, bør du selv bruke Swing for å unngå "blanding".
- ✓ *Utvikling over tid:* AWT bruker alltid seneste versjon av APIer som OS tilbyr og henger derfor med i utviklingen av "L&F" for hvert OS. Swing kan tenkes å emulere et "gammelt" utseende.
- ✓ *Leverandørstøtte:* Det hevdes at Sun/Oracle utvikler Swing mer aktivt enn AWT. Det kan se ut til at Swing er fremtiden.

Min konklusjon: Buk AWT bare til enkle applets og applikasjoner som bare skal brukes på én plattform. Bruk Swing ellers.

Hvis du senere skulle ønske å bruke AWT vil det ikke kreve noen særlig ekstra læring. *I dette kurset bruker jeg derfor kun Swing.*

Hva med SWT ("Standard Widget Toolkit")?

Dette biblioteket ble laget av IBM men vedlikeholdes nå av Eclipse. Det er en mellomting mellom AWT og Swing. SWT er "heavyweight" som AWT der det er mulig, men "lightweight" som Swing når det er påkrevet fordi det ikke finnes passende "peers" i det kjørende OS.

SWT utnytter altså farten i OS når det kan og emulerer (tregere) bare når det må. Det siste poenget gjør at det må brukes forskjellig SWT avhengig av plattform, noe som gjør SWT mindre portabelt. Mange mener at SWT skulle løse problemet med Swings eksekveringshastighet, men at det ikke lenger er så aktuelt – Swing blir raskere og raskere for hver versjon. Med lite minne og prosessorkapasitet (f.eks. på en mobil) kan det nok allikevel fortsatt være store forskjeller.

SWT hevdes å ha flere kontroller enn AWT, men færre en Swing. Det er også kritisert for å være for tett knyttet til Windows.

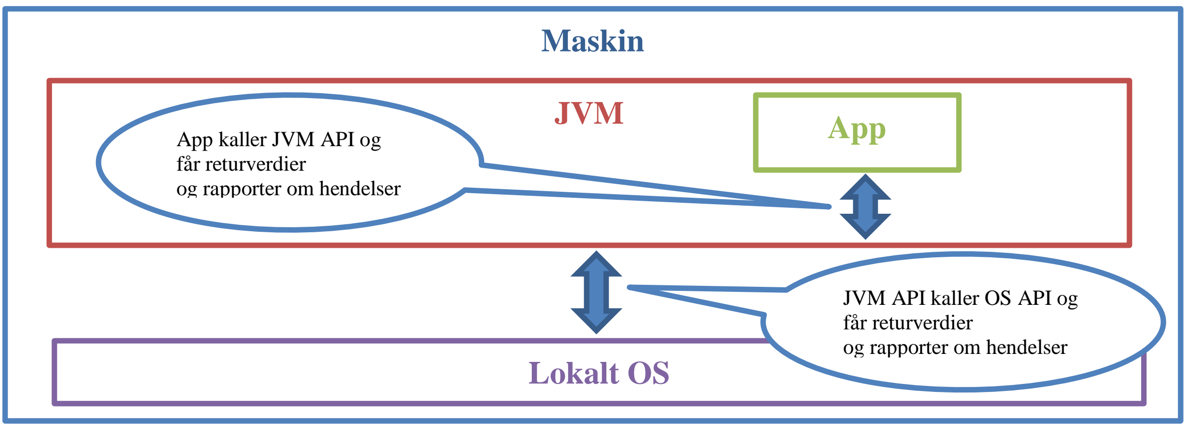
SWT kan ikke brukes med applets.

For meg ser det ikke spesielt interessant ut, og jeg tar det ikke opp i dette kurset.

Forholdet mellom JVM og OS

Java kjører i en virtuell maskin (JVM). Som virtuell maskin har den da et virtuelt operativsystem med en API. Den tilbyr da – som andre operativsystemer – metoder som vi kan bruke i våre Java-programmer (dvs. kompilatoren ordner det). Siden JVM kjører under et annet OS, f.eks. Windows, Linux, Mac OS, så kan den ikke selv ha kontakt med hardware. JVM må da henvende seg til det lokale OS API for å få ting gjort. Det er litt tungt, men fordelen er at når alle JVM har samme API, så kan vår app kjøre på alle maskiner. Det må imidlertid lages en egen JVM for hvert OS den skal kjøres under.

[2] Med NetBeans velger du "L&F" i prosjektets egenskaper.



Vår apps kall til JVM må "oversettes" til kall på OS API. Her kan det være at ett kall i Java API blir til flere kall i OS API og det kan kreves konvertering av datatyper og annet. Dette er en jobb som krever ressurser. Det krever dessuten at alle kall på JVM API faktisk kan oversettes til alle OS API. Det begrenser Java API til kall som lar seg oversette til alle lokale OS API. Kompilatoren og de som skrev koden for JVM for dette OS, ordner alt dette, så det får ingen betydning for vår kode.

Det viktigste for oss, er at OS API rapporterer hendelser til JVM. Det kan f.eks. være et klikk med venstre museknapp. Da inntreffer et *avbrudd* (interrupt) som OS oppfatter og straks håndterer. Det følger med et *Event*-objekt som beskriver hendelsen. JVM må så finne ut hvilken app hendelsen gjelder og rapportere hendelsen videre til den. Der må vi selv skrive kode som håndterer dem. Mer om dette nedenfor.

Lage grafisk brukergrensesnitt

For å få vist brukeren et vindu i en applikasjon (Applets tar vi ikke opp i dette kurset), må man bruke enten

1. *JFrame*
   Den gir et vanlig vindu, og det vil vi vanligvis ha i applikasjonen (minst ett)
2. *JDialog*
   Den gir et vindu for dialoger med brukeren (dialogbokser). Den har metoden *showDialog()* og er en funksjon som returnerer en verdi avhengig av brukerens svar. *JDialog* bruker vi som oftest ved å be et annet objekt/klasse – gjerne *JOptionPane* – om å lage det for oss. *JOptionPane* har mange metoder som kan lage mange forskjellige dialogbokser. Vi kan også selv designe dialogboksen helt fritt ved å utnytte arv fra *JDialog*.

Vi starter altså vanligvis med en *JFrame*. Det er en ferdig klasse, og følgelig kan vi ikke endre på dens metoder. Vi skal jo ha gjort noe når brukeren klikker på en knapp, fyller ut et tekstfelt osv. og da må vi få skrevet vår egen kode. Det ordner vi ved å lage vår egen klasse som *arver* fra *JFrame*.

```
public class GrafikkDemoView extends JFrame{
      private static GrafikkDemoView vindu = new GrafikkDemo();
}
```

I denne klassen kan vi da fritt legge til og overstyre/overlaste metodene som følger med fra *JFrame*. Legg merke til at vi har skapt et *static* objekt av denne klassen. For å unngå å lage egen programklasse (som egentlig ikke skal gjøre noe som helst annet enn å vise vårt skjema) lager vi en *main* i vår klasse.

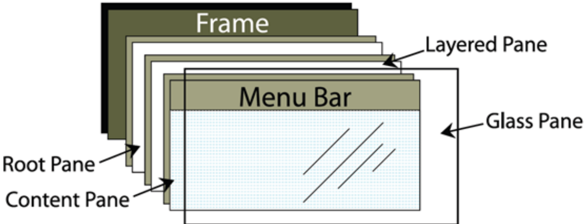
```
public static void main(String[] args) {
    vindu.init();
}
```
I *Main* kunne vi lagt all koden som skal generere vinduet med knapper, tekstbokser osv., men det har jeg delegert til metoden *init()*[3].

Alternativt kan man selvsagt lage en programklasse slik du antakelig er vant til, å skape/initiere skjemaet der:
```
public class GrafikkDemo {
    public static void main(String[] args){
        GrafikkDemoView skjema = new GrafikkDemoView();
        skjema.init();
    }
}
```
Nedenfor antar jeg at første teknikk er brukt – med *vindu* som *static* variabel.

Ekstra: Prinsipper for JFrame

Før jeg går videre, kan det være greit å få vite litt mer om hvordan grensesnittet bygges opp[4].

En *JFrame* er altså et vindu. Inn i dette vinduet, så vinduet fylles helt, legges automatisk en *RootPane*. *RootPane* har tre "lag" i form av *Panes* som ligger oppå hverandre. Det er ikke ofte vi har bruk for det, men det kan kanskje være greit å vite hvis man vil gjøre noe spesielt:



1. *Glass Pane* er vanligvis skjult men hvis det vises, så er det gjennomsiktig. Man kan tegne grafisk på den, og da synes grafikken hvis *Glass Pane* gjøres synlig.
2. *Content Pane* inneholder alle synlig komponenter unntatt evt. meny.
3. *Layered Pane* holder orden på hvor komponentene skal stå. En evt. menylinje legges her og da reduseres størrelsen av *Content Pane* tilsvarende. I tillegg inneholder den flere lag som jeg ikke går inn på her.

Vi bryr oss altså ikke så mye om dette, men oppfatter *JFrame* som ett, eneste objekt med komponenter på.

Bestemme LayoutManager og andre egenskaper ved vår JFrame

Når komponenter legges til en *JFrame*, så må de plasseres på et sted innenfor rammen. Plasseringen bestemmes av en *LayoutManager* som er en egenskap for *JFrame*. Den kan vi tilordne, sammen med andre egenskaper:

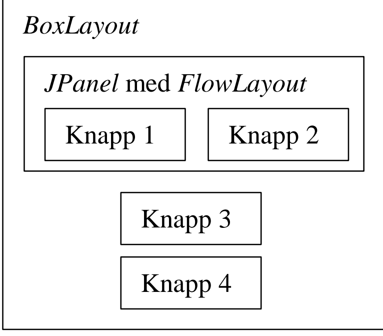
[3] Det vil også gjøre det enklere å lage Applet, men det tar vi ikke her.
[4] Deler av dette er hentet fra "Getting started with Java GUI Development" fra refcardz.dzone.com. De utgir mange nyttige, gratis "refcards" ("jukselapper").

```
public void init(){//objektmetode
    //Sett diverse egenskaper for vinduet
    setTitle("GrafikkDemo");
    setLayout(new FlowLayout(FlowLayout.CENTER,0,50));
    setSize(500,300);
    setDefaultCloseOperation
         (JFrame.DISPOSE_ON_CLOSE); //hva skjer ved lukking
```

Her endrer jeg *LayoutManager* fra *BorderLayout* (som er default) til *FlowLayout*)[5]. Videre setter jeg tittel, størrelse og hva som skal skje når brukeren lukker vinduet (default er *HIDE_ON_CLOSE* som bare skjuler vinduet).

FlowLayout lages her som "anonymt objekt". Det kan passe når vi ikke skal referere til det etterpå. Ulempen er at det kompileres en egen fil som klasse for dette objektet. Det heter *GrafikkDemo$1.class* for den første, *$2* for den andre osv. Alle *class*-filene må være med hvis den kompilerte koden skal flyttes til en annen katalog. (Det gjøres vanligvis ved at alle kompilerte klasser legges i en *jar*-fil.)

Det kan være nødvendig å nøste *LayoutManager* (legge en inne i en annen) for å få det utseende du ønsker. Da legger du inn komponenten *JPanel* på passende sted og setter passende *LayoutManager* for den. Hvis vi f.eks. har en *BoxLayout* med bare én kolonne, og vil ha to knapper ved siden av hverandre øverst, kan vi legge inn en *JPanel* der med en annen layout:



Vi har her lagt til et *JPanel* og to knapper i *BoxLayout*, deretter har vi lagt til to knapper i *JPanel*.

Legge til komponenter

Man vil også sette egenskaper for komponentene. De er skapt som objektattributter:

```
public class GrafikkDemoView extends JFrame{
    private static GrafikkDemoView vindu = new GrafikkDemoView();
    private JButton butHei = new JButton("Si hei!");
    private JTextField txtNavn = new JTextField();
    private JLabel lblNavn = new JLabel("Oppgi navnet ditt: ");
```

Egenskapene for kontrollene settes i *init()*:

```
        //Sette egenskaper for komponentene
        txtNavn.setColumns(20);
        butHei.setEnabled(false);
```

Deretter må de legges til *JFrame* (selv om de er felter der – de skal nå legges til i *RootPane*):

```
        //Legg til komponentene
        add(lblNavn);
        add(txtNavn);
        add(butHei);
```

Skjemaet vårt har nå fått komponenter og de er plassert på skjemaet av *FlowLayout*.

---

[5] Andre LayoutManagers er vist senere i kapittelet, side 14.

Alt er nå på plass i skjemaet og det kan gjøres synlig (default er at det er usynlig):

```
            setVisible(true);
```

Lyttere

Hendelsene som ankommer til vår applikasjon skal (delvis) føre til handlinger. Da må vi lage en eller flere lytter som fanger opp hendelsene og håndterer dem.

Man kan la skjemaet selv, *JFrame*, fange opp alle hendelser, men da må man skrive kode som sjekker hvilken hendelse som faktisk inntraff og starte riktig handling deretter. Jeg synes det er enklere å lage en lytter for hver komponent som skal reagere, så slipper jeg å sjekke så nøye hva som faktisk hendte.

Hvis lytterne bare skal være for én komponent hver, er det ikke nødvendig å deklarere en variabel. Isteden legger man lytteren til "anonymt". Lytteren legges til komponentene som skal reagere:

```
   //Legg til lyttere
   butHei.addActionListener(new ButStartLytter());
   txtNavn.addKeyListener(new TxtNavnLytter());
```

Hvis en lytter skal legges til flere komponenter, må den skapes som variabel først:

```
   //Instansier lytteren
   ButStartLytter butLytter = new ButStartLytter();
   TxtNavnLytter navnlytter = new TxtNavnLytter();
   //Legg til lyttere
   butHei.addActionListener(butLytter);
   txtNavn.addKeyListener(navnlytter);
```

Lytterne er mine egne, indre klasser som implementerer et passende grensesnitt:

```
   private class ButStartLytter implements ActionListener{
      [6]public void actionPerformed(ActionEvent e){
         JOptionPane.showMessageDialog
                 (rootPane, "Hei, "+ txtNavn.getText() + "!");
      }
   }
```

ActionListener er et grensesnitt og krever bare at metoden *actionPerformed* defineres.

```
   private class TxtNavnLytter implements KeyListener{
      //Må definere tre operasjoner
      @Override public void keyReleased(KeyEvent k){
         butHei.setEnabled(!txtNavn.getText().isEmpty());
      }
      public void keyPressed(KeyEvent k){ }
      public void keyTyped(KeyEvent k){ }
   }
```

KeyListener er også et grensesnitt. Den krever at tre metoder defineres. Her gir jeg kode bare til den ene, da de andre er unødvendig i denne applikasjonen, men de må overstyres.

Indre klasser kompileres til egne filer. Den første får her navnet *GrafikkDemoView$ButStartLytter.class*. Også disse må være med hvis den kompilerte koden flyttes.

[6] Avhengig av programmeringsmiljøet du bruker, kan det hende at du foreslåes å legge til *@Override* her, siden *actionPerformed* er en arvet metode som implementeres her. Denne *@Override* er en kommando til kompilatoren om at du mener dette som en overstyring av arv eller implementering av grensesnitt og den skal gi feilmelding hvis den ikke finner noe som kan overstyres/implementeres – da har du gjort en feil. Den gjør det altså tryggere, men er ikke strengt nødvendig.

Merk at *keyTyped* inntreffer før tegnet er sendt til textfeltet og derfor vil *isEmpty()* fortsatt være true når første tegn er skrevet. Dette er bevisst gjort, iflg feilloggen for Java:

> EVALUATION
> The listeners are notified of the key events prior to processing them to allow the listeners to "steal" the events by consuming them. This gives compatibility with the older awt notion of consuming events. The "typed" event does not mean text was entered into the component. This is NOT a bug, it is intended behavior.
> timothy.prinzing@eng 1998-05-22

KeyReleased, derimot, fyres av etter at tegnet er kommet frem.

De to klassene implementerer forskjellige grensesnitt og derfor også forskjellige metoder. Grunnen til at vi implementerer forskjellige grensesnitt, er delvis at kontrollene reagerer på forskjellige typer hendelser og delvis at det er hendelser vi ikke ønsker å håndtere. F.eks. vil en label ikke reagere på en *actionperformed*-hendelse. Et annet eksempel er at hvis brukeren klikker i en tekstboks, så vil vi ikke gjøre noe, selv om det genereres en klikk-hendelse for den.

Det kan være vanskelig å velge riktig type *EventListener*[7]. Det er et meget stort antall forskjellige grensesnitt som arver fra den:

java.util

Interface EventListener

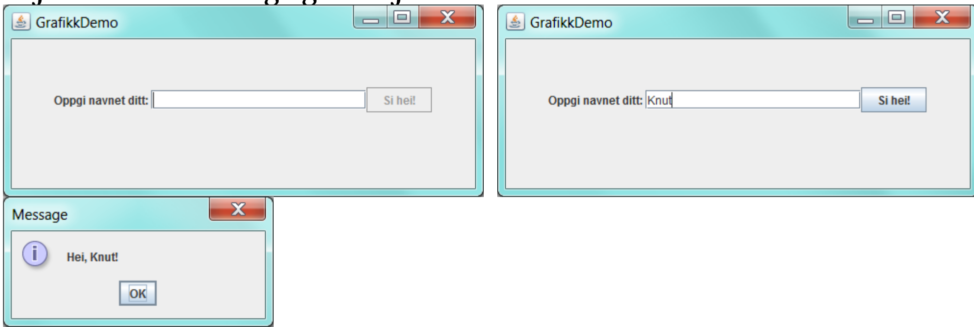
All Known Subinterfaces:

Action, ActionListener, AdjustmentListener, AncestorListener, AWTEventListener, BeanContextMembershipListener, BeanContextServiceRevokedListener, BeanContextServices, BeanContextServicesListener, CaretListener, CellEditorListener, ChangeListener, ComponentListener, ConnectionEventListener, ContainerListener, ControllerEventListener, DocumentListener, DragGestureListener, DragSourceListener, DragSourceMotionListener, DropTargetListener, FlavorListener, FocusListener, HandshakeCompletedListener, HierarchyBoundsListener, HierarchyListener, HyperlinkListener, IIOReadProgressListener, IIOReadUpdateListener, IIOReadWarningListener, IIOWriteProgressListener, IIOWriteWarningListener, InputMethodListener, InternalFrameListener, ItemListener, KeyListener, LineListener, ListDataListener, ListSelectionListener, MenuDragMouseListener, MenuKeyListener, MenuListener, MetaEventListener, MouseInputListener, MouseListener, MouseMotionListener, MouseWheelListener, NamespaceChangeListener, NamingListener, NodeChangeListener, NotificationListener, ObjectChangeListener, PopupMenuListener, PreferenceChangeListener, PropertyChangeListener, RowSetListener, RowSorterListener, SSLSessionBindingListener, StatementEventListener, TableColumnModelListener, TableModelListener, TextListener, TreeExpansionListener, TreeModelListener, TreeSelectionListener, TreeWillExpandListener, UndoableEditListener, UnsolicitedNotificationListener, VetoableChangeListener, WindowFocusListener, WindowListener, WindowStateListener

I praksis vil dere mest bruke *ActionListener*, *KeyListener* og *MouseListener*.

Oppsummering

Skjemaet er nå ferdig og kan kjøres. Det vil se omtrent slik ut:



Forskjellen på de to er at til høyre er det skrevet noe i tekstboksen, og da er knappen virksom. Det er *txtNavnLytter* som ordner det. Klikk på knappen gir en meldingsboks – det ordnes av *butStartLytter*.

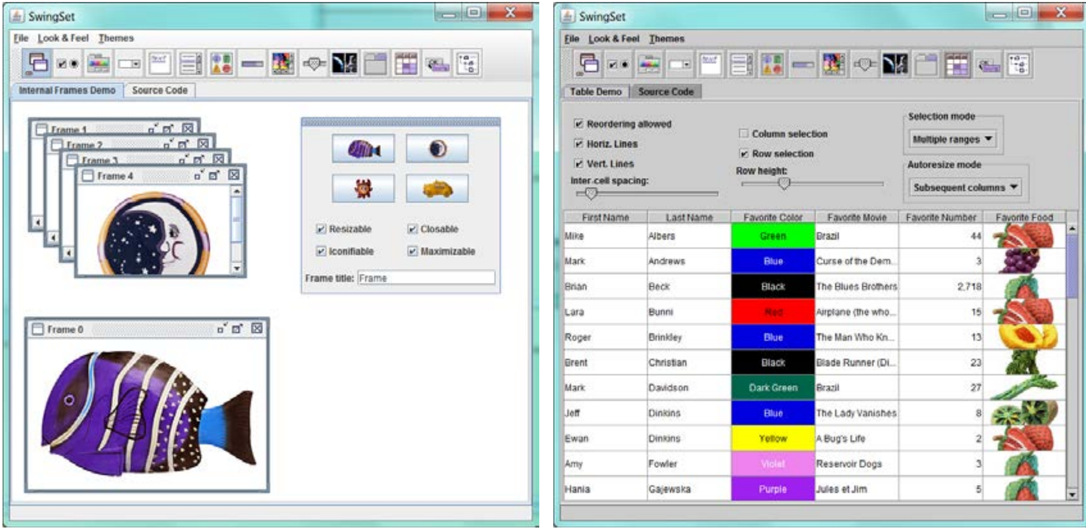
[7] Du finner en fin oversikt over hva de forskjellige hendelsene rapporterer på http://www.roseindia.net/java/example/java/awt/Awt_Events.shtml

Oppskrift:

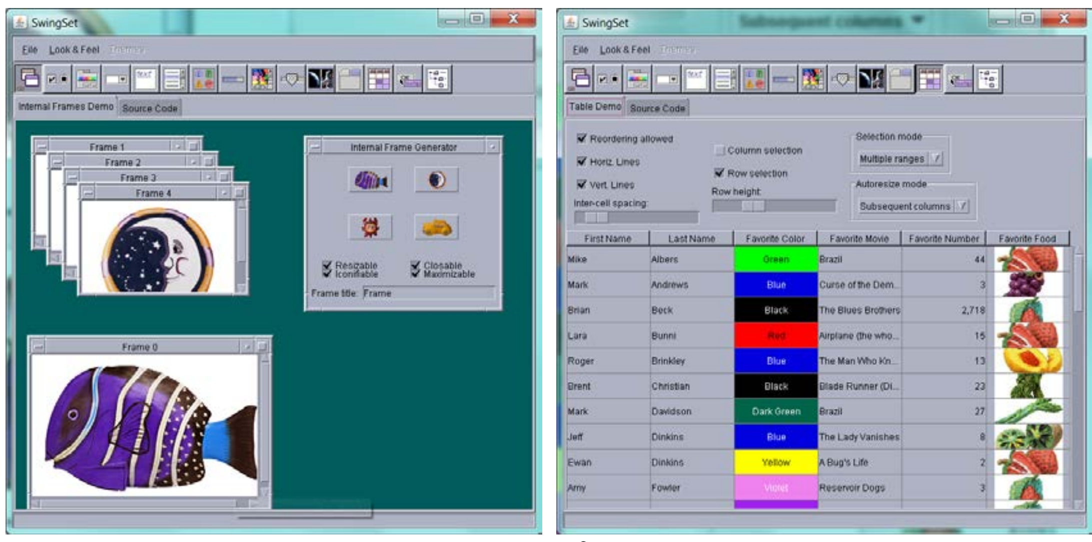
1) Importer *java.awt.**, *javax.swing.** og *java.awt.event.** og annet etter behov.
2) Lag programklassen som subklasse til (med arv fra) *JFrame*.
3) Lag en statisk variabel for skjemaet og instansier den.
4) Legg til alle komponentene som objektvariable i programklassen og instansier dem (instansieringen kan også gjøres i init-metoden).
5) Lag en statisk metode *main* som vanlig for programklasser. I den kaller du bare skjemaets *init()*.
6) Lag en metode *init()* der du gjør følgende:
   a) Sett egenskaper for skjemaet.
   b) Sett egenskaper for komponentene.
   c) Lyttere:
      i) Instansier og legg til anonyme lytter til hver kontroll, eller
      ii) Instansier lyttere som variable og legg lytterne til kontrollene
   d) Legg komponentene til skjemaet
   e) Gjør skjemaet synlig (default er at det er usynlig).
7) Deklarer/definer lytterklassene for hver kontroll, som indre klasse i programklassen (det er betydelig vanskeligere å lage en ytre lytterklasse). Lytterklassen må implementere en *EventListener*.

Det fremgår vel at dette er ganske tungt for programmereren. Dette gjør vi da heller ikke i praksis – vi bruker et programmeringsmiljø med grafisk grensesnitt. (Vi skal bruke NetBeans i dette kurset.) Imidlertid må man allikevel forstå hva som skjer, slik at man kan gjøre fornuftige valg i programmeringsmiljøet. Fra tid til annen er det også nødvendig – eller enklere – å "hjelpe" programmeringsmiljøet litt.

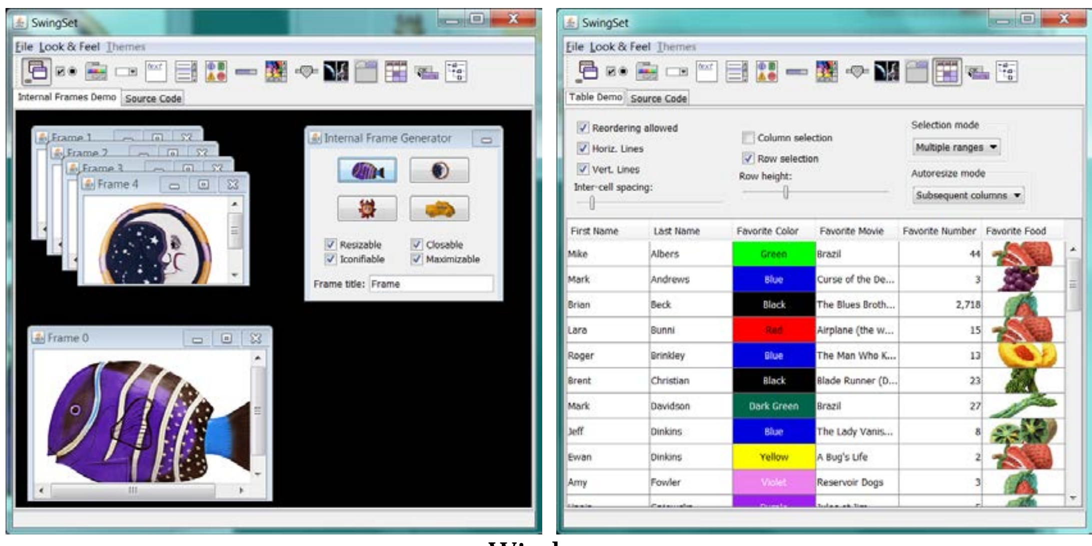
Ekstra: Swing "L&F" Bilder



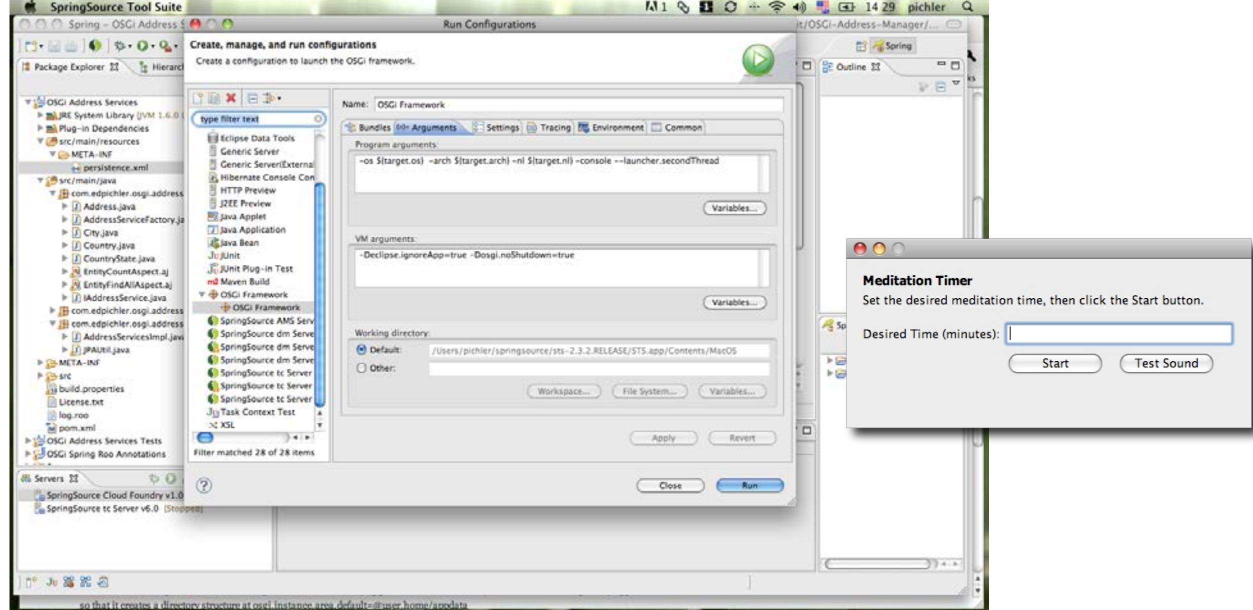
Java default "Metal" (har også mange "themes")



Motif



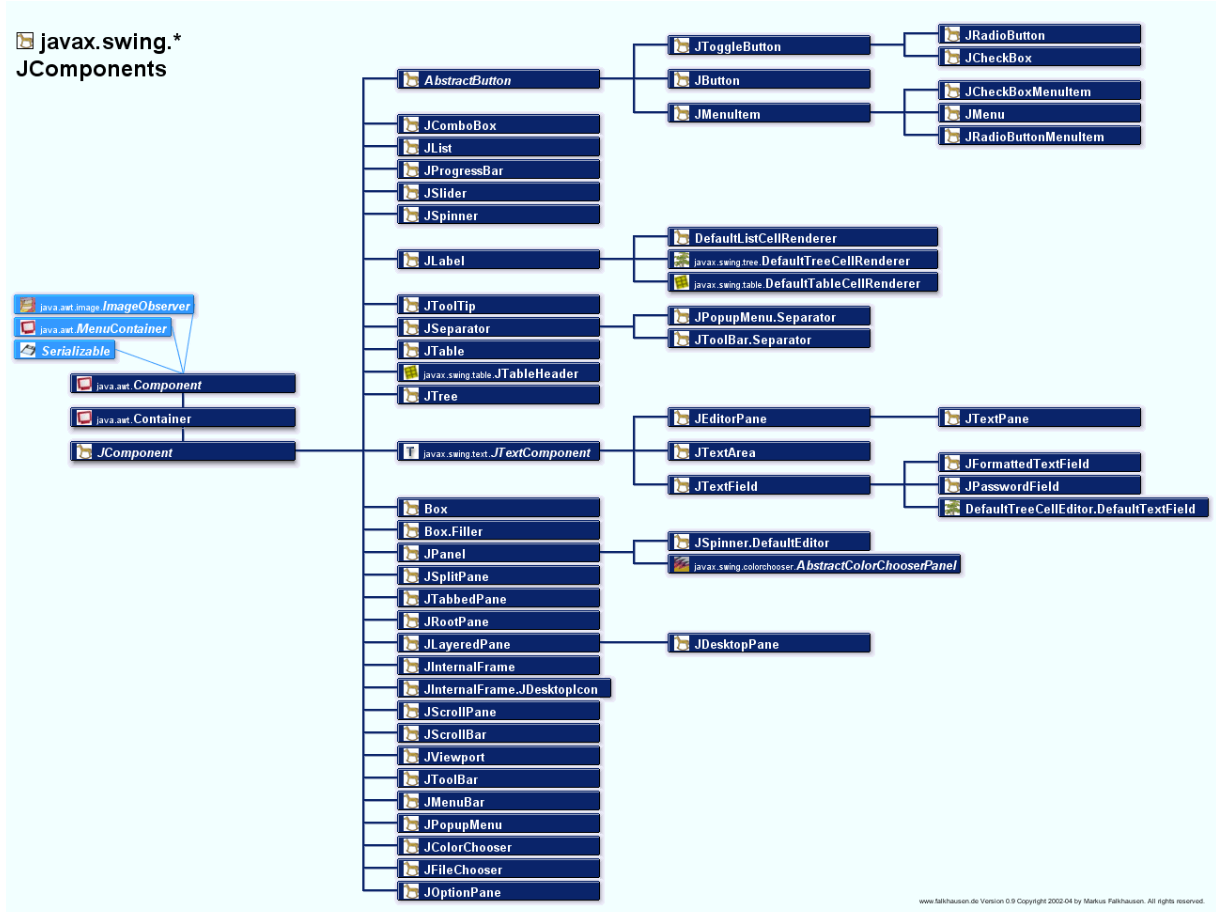
Windows



Mac OS

Ekstra: Oversikt over Swing

Hoveddeler i arvehierarkiet for javax.swing



Figuren er laget av Markus Falkhausen og er interaktiv på nett. Se
http://www.falkhausen.de/en/diagram/html/javax.swing.JComponents.html

Kort beskrivelse for noen av komponentene

Component	Kort beskrivelse
Box.Filler	En usynlig komponent som kan brukes for å skille andre komponenter fra hverandre o.l.
ButtonGroup	Knytter flere knapper sammen, slik at bare én kan være trykket inn av gangen, typisk *JRadioButton* brukt til å angi valg. (Knappene kan teoretisk være av hvilken som helst subklasse til *AbstractButton*.)
JButton	Kommandoknapp.
JCheckBox	Sjekkboks.
JComboBox	PopUp meny. Det som er valgt, vises i feltet.
JLabel	En linje med output-tekst.
JList	En liste med valg.
JMenu	Vises som meny enten øverst i vinduet (i en *JMenuBar*) eller som pop up meny (i en *JPopupMenu*).
JProgressBar	Viser hvor langt en jobb som tar tid, er kommet.
JRadioButton	Radioknapp. Brukes ofte sammen med andre radioknapper i en *ButtonGroup*.
JScrollBar	Gir brukeren mulighet til å velge en verdi langs en kontinuerlig skala (f.eks. fargemetning) ved å "skyve en glider" opp/ned eller høyre/venstre.
JSpinner	Et lite opp/ned ikon som gir brukeren mulighet til å velge mellom flere verdier (forrige/neste).
JTextArea	Flere linjer med inputtekst (rows og columns) og kan ha scrollbars..
JTextField	En linje med inputtekst.

Noen containers (klasser – må arves)

Container	Kort Beskrivelse
JApplet	Kjøres under annet program, f.eks. AppletViewer eller webleser. Applet har verken borders eller title (i en webleser vises tittelen på websiden og i AppletViewer vises klassenavnet).
JDialog	Et dialogvindu.
JFrame	**Det typiske Java-vinduet** med borders og title. Kontrollene plasseres i *Content pane* eller *menu* bar. Instansieres og vises i grafiske applikasjoner.
JPanel	Enkelt område uten borders for gruppering av komponenter/containere.
JScrollPane	Har automatisk rullefelt for den ene komponenten som inkluderes, f.eks. en tekstfil.
JWindow	Dette vinduet har ikke borders og title, men er et separat vindu.

Noen eventlisteners (interfaces – må implementeres og metodene må defineres)

Både *JApplet* og *JFrame* kan implementere *ActionListener* og må da definere metoden *public void actionPerformed(ActionEvent e)*.

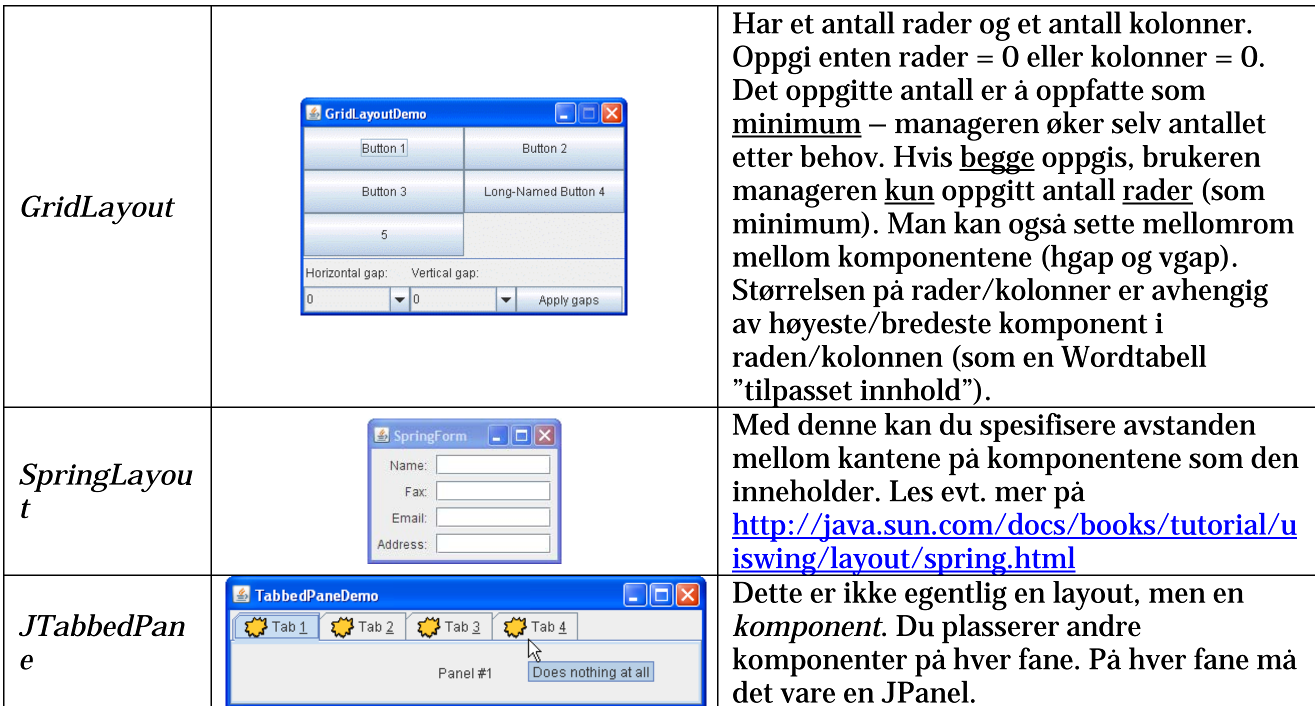
Følgende kontroller kan tilknyttes en *ActionListener*: *JButton*, *JCheckBox*, *JComboBox*, *JFileChooser*, *JFormattedTextField*, *JMenuItem*, *JPasswordField*, *JRadioButton*, *JTextField* og *JToggleButton*. Alle Swingkomponenter kan tilknyttes disse lytterne:

ComponentListener, *FocusListener*, *KeyListener*, *MouseListener*, *MouseMotionListener*, *MouseWheelListener*, *HierarchyListener* og *HierarchyBoundsListener*.

Listener	Kort beskrivelse
ActionListener (interface)	**Metode (må defineres):** *public void actionPerformed(ActionEvent e)* som reagerer når brukeren vil ha reaksjon, f.eks. klikket på en knapp eller trykket ENTER i et tekstfelt. **ActionEvent:** Med *String getActionCommand()* kan man finne hvilken container/component som ble utsatt for hendelsen. Det er default teksten på knappen/sjekkboksen osv. blir returnert, hvilket er OK for knapper o.l., men lite brukbart for tekstfelt. Med *setActionCommand(<tekst>)* kan du bestemme hva *getActionCommand()* skal returnere. Alternativt vil *Object getSource()* vise hvilken kontroll hendelsen skjedde med. Med *int getModifiers()* kan man sjekke SHIFT o.l. (returnerer alle kombinert med logisk OR: SHIFT=1, CTRL=2, META=4, ALT=8 osv.).
MouseListener (interface) eller MouseInputAdapter (abstract klasse)	**Metoder (må defineres):** *public void mouseClicked(MouseEvent e)*, *public void mouseEntered(MouseEvent e)*, *public void mouseExited(MouseEvent e)*, *public void mousePressed(MouseEvent e)* og *public void mouseReleased(MouseEvent e)*. Reagerer når brukeren anvender musen på/over en kontroll. **Alternativ:** *public abstract class MouseInputAdapter* der ovennevnte metoder (og noen flere) er definert tomme – lag egen subklasse og overstyr de metodene som er aktuelle. **MouseEvent:** Med *boolean isPopupTrigger()* kan man sjekke om det var (vanligvis) høyre musetast som ble klikket og *InputEvent* har metoder for å sjekke SHIFT osv. og tidspunktet.
KeyListener (interface)	**Metoder (må defineres):** *public void keyPressed(KeyEvent e), public void KeyReleased(KeyEvent e) og public void KeyTyped(KeyEvent e)* som reagerer når brukeren anvender tastaturet. *KeyTyped* fyres av *før* komponenten har mottatt tegnet så man kan endre det. **KeyEvent:** Bare for *KeyTyped* vil *char getKeyChar()* returnerer tegnet, f.eks. 'a' eller 'A'. *int getKeyCode()* returnerer en kode for tasten. *public int getKeyModifiers()* returner en "maske" for hvilke andre taster (CTRL, ALT osv. som ble holdt nede samtidig). Du kan endre tegnet før det sendes videre til komponenten med *void setKeyChar(char keyChar)* og andre.

Noen layoutmanagers
Ref: http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html

Manager	Bilde	Kort beskrivelse
GroupLayout **Denne bruker vi mye, men bare med NetBeans**		Ganske kompleks layout med to, separate lag – vertikalt og horisontalt. Den plasserer komponentene akkurat der du vil ha dem. Sun anbefaler at den kun brukes med grafiske designverktøy og den er standard for JPanel og JFrame i NetBeans.
BorderLayout		Containeren deles i fem regioner: PAGE_START, PAGE_END, LINE_START, CENTER og LINE_END. Angi region ved innsettingen: add(<komponent>,<region>) der region er *BordeLayout.CENTER* osv. Regionene øverst og nederst vil ha prioritet over linjen tversover.
BoxLayout		Har enten bare én rad eller én kolonne, ellers nokså lik *GridLayout*.
CardLayout		Bare én komponent (vanligvis en *Panel*) vises av gangen. De andre ligger under som i en kortstokk.
FlowLayout		Fyller på med komponenter venstre=>høyre, ovenfra=>ned omtrent som tekst med word wrap. Radhøyden bestemmes av den høyeste komponenten på linjen og størrelsen på komponentene bestemmer manageren ("naturlig"/preferred). Dette er default layout for alle JPanels.
GridBag-Layout		Mer fleksibel form for *GridLayout* som likner på HTML-tabeller. Kontroller kan gå over flere celler. Man kan angi både antall rader og antall kolonner en komponent skal "fylle". En komponent legges til med addLayoutComponent(<component>,<constraint>) der constraint er av klassen *GridBagConstraint* med egenskapene gridwidth og gridheight satt til et passende heltall (de er public int og kan settes direkte).



GridLayout		Har et antall rader og et antall kolonner. Oppgi enten rader = 0 eller kolonner = 0. Det oppgitte antall er å oppfatte som <u>minimum</u> – manageren øker selv antallet etter behov. Hvis <u>begge</u> oppgis, brukeren manageren <u>kun</u> oppgitt antall <u>rader</u> (som minimum). Man kan også sette mellomrom mellom komponentene (hgap og vgap). Størrelsen på rader/kolonner er avhengig av høyeste/bredeste komponent i raden/kolonnen (som en Wordtabell "tilpasset innhold").
SpringLayout		Med denne kan du spesifisere avstanden mellom kantene på komponentene som den inneholder. Les evt. mer på http://java.sun.com/docs/books/tutorial/uiswing/layout/spring.html
JTabbedPane		Dette er ikke egentlig en layout, men en *komponent*. Du plasserer andre komponenter på hver fane. På hver fane må det vare en JPanel.

Oppgave til kapittel 1

Læringsmål

Det er viktig å forstå hvordan Java håndterer vinduer og grafikk, selv om NetBeans og andre verktøy gjør det meste for oss. Vi må jo allikevel velge kontroller, sette egenskaper og bruke dem.

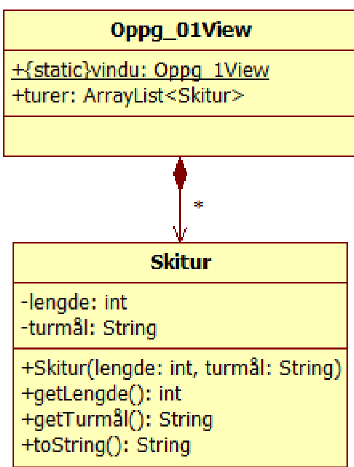
Videre er det svært nyttig å få trening med å slå opp i JavaDoc. Selv meget erfarne programmerere (og der er ikke dere riktig ennå!) trenger å slå opp – det er alt for mye å huske og det endrer seg også fra tid til annen.

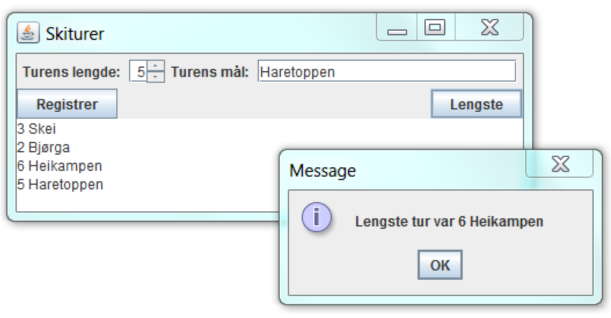
Denne oppgaven tar sikte på å trene begge deler. Dere skal derfor bruke en teksteditor og ikke *NetBeans* o.l. som kommer med forslag til rettelser, metoder osv. Det er allikevel greit å bruke en editor som farger ord, lager ryddig kode osv. Det er siste og eneste sjanse dere får til å trene grafikkbruk på denne måten – senere bruker vi NetBeans.

Oppgave

Det skal lages et grafisk brukergrensesnitt som nedenfor, der det er mulig å lagre opplysninger om skiturer. Det skal ikke lagres noe på fil/database denne gang – hensikten er å trene grensesnittet. Vi lager heller ikke noen kontrollklasse.

Klasse og skjemaer:





Du skal…

1. *kun benytte en teksteditor* (og ikke et integrert IDE som f.eks. NetBeans). Du skal lage alt manuelt og få trening i å slå opp i dokumentasjonen.
2. lage klassen *Skitur*.
3. lage vindu med *JFrame* og de angitte kontrollene.
4. bruke *BorderLayout* til å plassere kontrollene – i det øverste feltet legger du en *JPanel* (som har *FlowLayout* som standard) slik at du kan få flere kontroller inn der.
5. lagre dataene midlertidig i en *ArrayList* etter hvert som de registreres.
6. vise dataene i en *JTextArea* i den rekkefølgen de ble lagt inn.
7. vise den lengste turen som er registrert i en *JDialog* (bruk gjerne *JOptionPane*).

Kapittel 2 – NetBeans

Jeg omtaler her NetBeans versjon 8.0 med Swing. I ditt kurs har du antakelig en nyere versjon av NetBeans, men forskjellene er neppe store.

Noen demovideoer om NetBeans

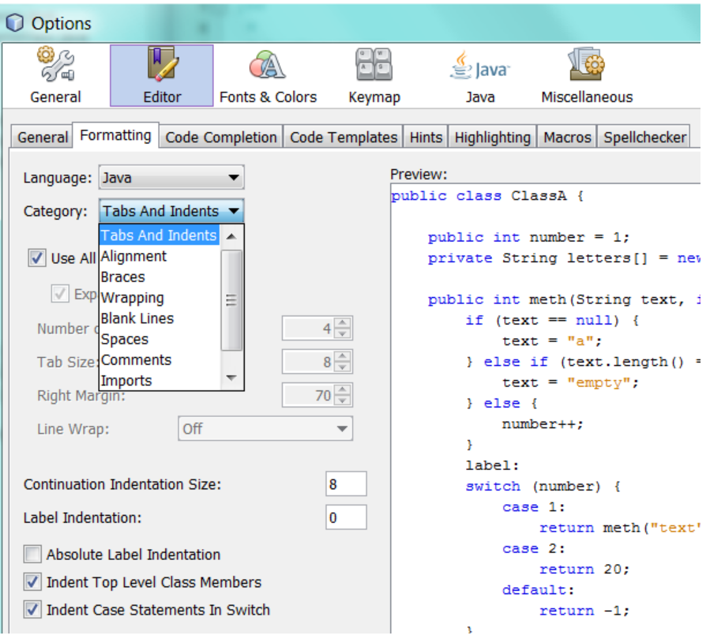
Jeg har laget noen videoer om bruken av NetBeans som du kanskje vil se på etter at du har fått installert programmet[8]. **"NetBeans Intro"** viser hvordan vi starter med å lage et program med NetBeans. **"NetBeans klasser"** viser hvordan NetBeans kan hjelpe til med å lage dine egne klasser.

Installere og tilpasse NetBeans

Man kan selvsagt skrive Java-kode i en hvilken som helst editor, men som vi har erfart er det tungt å få til et grafisk brukergrensesnitt på den måten. Vi har også tidligere erfart hvor mye assistanse en programmerer kan få av en god IDE (Integrated Development Editor). I resten av dette kurset skal vi bruke NetBeans[9].

Jeg antar at dere allerede har installert Java JDK 1.8 (ofte kalt bare "Java 8") med seneste update. Hvis ikke, må dere installere den *før* NetBeans. Vi bruker "Java SE 8" (der "SE" står for "Standard Edition"). Dere installerer deretter enkelt NetBeans i seneste produksjonsversjon (dere bør unngå beta-versjoner). Selv har jeg NetBeans IDE 8.0. *Pass på å få med JUnit som vi skal benytte i dette kurset.* Sjekk etter installasjonen at NetBeans er tilknyttet Java 1.8 – enklest under *Help|About*.

Når NetBeans er installert, kan dere sette opp noen standarder for prosjektene. Du finner dem under *Tools|Options*.



Velg "Java" og sett opp innrykk, parenteser osv. slik du liker det. Selv liker jeg

1. linjeskift foran *else*, *while*, *catch* og *finally*
2. parenteser på samme linje (bak ordet)
3. å importere som pakker (dvs. med stjerne og ikke én og én klasse)
4. å ha mest mulig "pop up" under "code completion"

[8] Faglæreren din har tilgang til dem i en egen lærerressurs.
[9] Dere har tidligere kanskje brukt Eclipse, men jeg holder en knapp på NetBeans og synes generelt det er fint at dere får prøve flere.

5. å ha full kontroll over deling av linjer selv og har derfor skrudd av all automatisk "wrapping"
6. en annen farge enn standard for kommentarene, for på min PC er de nesten usynlige. Jeg har valgt fargen [0,102,102] som er en mørk blågrønn farge og gjort skriften fet
7. å ha *Code Templates* slik at *trycatch* og *ifelse* blir slik jeg ønsker[10]
8. å slippe teksten øverst som forteller meg hvordan jeg kan endre templates, derfor sletter jeg all kode og tekst i *default license template*
9. å skru av stavekontrollen

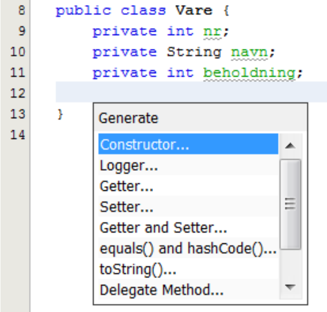
Du bestemmer selv hvordan *du* vil ha det.

Tips ved bruk av NetBeans

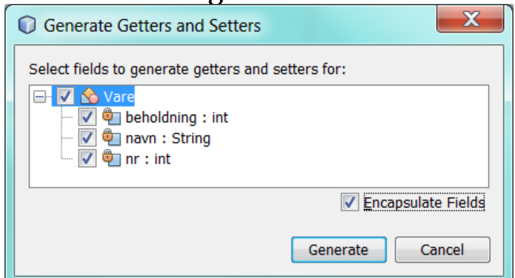
NetBeans gir mye hjelp til programmereren. Her er noen tips:

1. Når du skal gi noe nytt navn, så bruk *alltid* høyreklikk og velg *Rename*, *Refactor|Rename*[11] eller lignende. Navnet er ofte brukt mange steder, f.eks. både som klassenavn, konstruktørnavn og i kall, kommentarer o.a. NetBeans vil rette alle – selv vil du sikkert glemme noen.

2. Når du lager en ny klasse, kan du få laget konstruktør, toString, get- og set-metoder osv. Skriv først inn feltene (de variable). Plasser markøren der du vil ha satt inn kode og høyreklikk og velg *Insert code...* Velg hva du vil ha satt inn.



Ved noen av valgene får du liste du kan velge i:



Her velger du hvilke felt som skal være med, og kan be om at feltene merkes *private* ("*Encapsulate*").

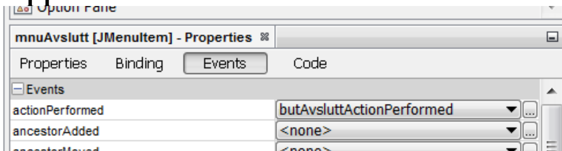
Tenk deg om for *equals* – når oppfattes to objekter som like? Ofte er det når de har like identifikatorer. *HashCode* må bruke de samme feltene som *equals* for det er et krav at to like objekter også skal ha samme hash-tall.

[10] Jeg husker ikke om noen av dem var med som standard☺!

[11] *Refactor* = Endre koden uten å endre kodens oppførsel. Det finnes egne verktøy som kan gjøre slikt og NetBeans har ganske mange muligheter utover bare *rename*. Hensikten er å fjerne *code smells* dvs. kode som funksjonerer korrekt men som er vanskelig å forstå og kan indikere dypere designproblemer. De kan forårsake feil ved senere vedlikehold og bør endres uten at funksjonaliteten endres. WikiPedia har en fin liste der du sikkert vil kjenne igjen mye. Interesserte anbefales også en titt på DZones refcardz om refactoring.

3. Metodene må redigeres litt etterpå. F.eks. mener jeg
    a. Konstruktøren bør benytte set-metoder slik at kontroller som er lagt i set-metoden også gjennomføres for konstruktøren.
    b. Set-metoder for identifikatorer bør være *private* så ikke identifikatoren kan endres.
    c. Set-metodene bør kontrollere verdien. Da vil de vanligvis kaste feil og må merkes *throws Exception*. Det vil da også gjelde konstruktøren som kaster feilen videre.
    d. *ToString* er svært enkel – du vil nok ha en bedre returstreng.
    e. Hvis objektene skal sorteres, må du lage *compareTo* selv[12].

4. Når flere komponenter skal reagere likt, f.eks. en meny og en knapp, kan du henvise til samme hendelse i *Events* eller kalle den ene fra den andre. Her har jeg satt *actionPerformed* for menyvalget *mnuAvslutt* til å bruke samme metode som knappen *butAvslutt*:



5. Når du prøvekjører, kompileres programmet. Hvis kompileringen gir feil, får du lite informasjon. Bruk da heller menyen *Run/ Build Project* som gir bedre feilmeldinger. Den finnes også i toolbar som en hammer.

6. Hvis du vil sette stoppunkter i programmet, klikker du til venstre for koden. For å få eksekveringen til å stoppe der, må du ikke bruke vanlig prøvekjøring, men bruke knappen *Debug*.

7. Hvis du får en melding om at en Java-klasse ikke er deklarert, har du antakelig en manglende *import*. Du kan få lagt til dem som mangler med høyreklikk i koden og *Fix imports*. NetBeans vil legge til dem som mangler, og fjerne dem som ikke er i bruk. Ofte vil du få valg mens du koder og importen lages automatisk. Uansett kan det være greit å "fix imports" til slutt så du ikke importerer unødvendige ting (selv om det ikke tar ekstra plass i den kompilerte bytekoden, så forvirrer det).

8. Når markøren er i et ord som finnes i JavaDoc, kan du høyreklikke og be om å få se JavaDoc. Du kommer direkte til riktig dokument i en nettleser.

---

[12] I tidligere Java-kurs lærte du å lage CompareTo. Her følger en liten repetisjon.
Slik lager du compareTo enkelt: La klassen implementere *Comparable<Venn>*. Hvis sammenlikningen baseres på et *objekt* (inkludert strenger) kan du skrive slik:

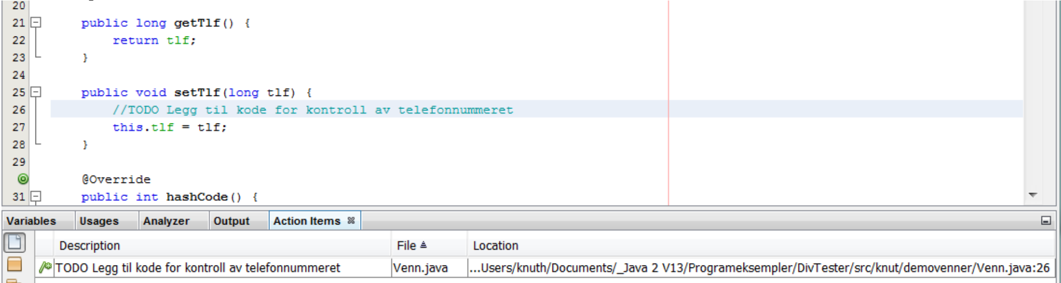
```
@Override
public int compareTo(Venn venn){
    return this.getNavn().compareTo(venn.getNavn());
}
```

Hvis identifikatoren derimot er en *primitiv* variabel må du bruke wrapper-klassens *compare*, f.eks.

```
@Override
public int compareTo(Venn venn){
    return Long.compare(this.getTlf(),venn.getTlf());
}
```

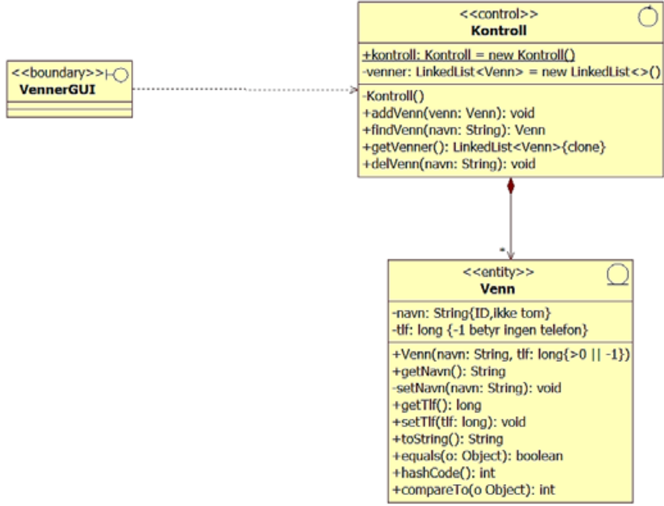
Hvis flere verdier skal sammenliknes, bruker du naturligvis et logisk uttrykk.
@Override er en annotering, dvs. en "beskjed" til kompilatoren, om at du tenker å overlaste en eksisterende, arvet metode. Du får feilmelding hvis det ikke er tilfelle. Annoteringen er ikke nødvendig, men gjør kodingen tryggere. Merk at vi her bruker *Venn* som parameter (fordi vi implementerer *Comparable<Venn>*) mens *equals* må ha *Object* som parameter fordi metoden er arvet fra klassen *Object*. *CompareTo* må returnere 0 hvis objektene er like i henhold til *equals*.

9. Bruk *//TODO* aktivt. I et vindu nederst ser du dem listet opp, og kan dobbelt-klikke på dem for å komme dit i koden:



Bruke NetBeans

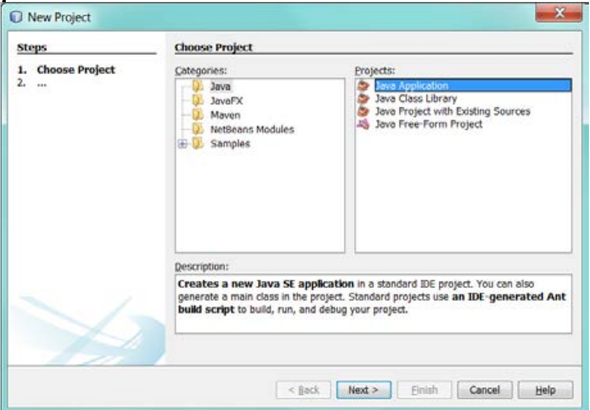
Vi skal ta utgangspunkt i en forenkling av den første oppgaven dere løste i OOAD-kurset. Klassediagrammet ser da slik ut:



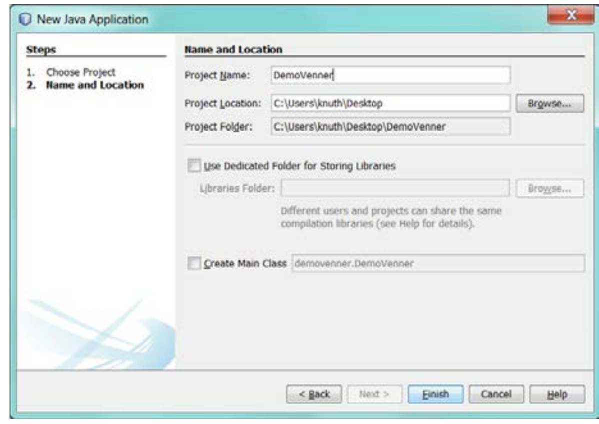
1. Skap et nytt *prosjekt*

Som i Visual Studio er programmer kalt "prosjekt". Et prosjekt vil bestå av en rekke filer som til sammen utgjør det kjørbare programmet. Mange av dem genereres av NetBeans.

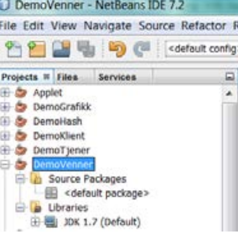
Du skal lage en *Java Application*:



Pass på i trinn to at du *ikke* har valgt *Use Dedicated Folder* eller *Create Main Class*. Velg et fornuftig prosjektnavn og lagringssted. Her vil NetBeans lage en submappe "DemoVenner" i Desktop-mappen. Alle prosjektets filer legges der.



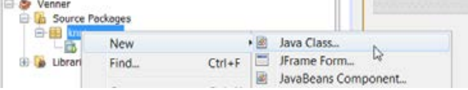
Når NetBeans har gjort sitt, har du fått en ny mappe "DemoVenner" i ditt filsystem, og du kan se prosjektet i vinduet til venstre:



2. Skriv kode for klassene

For å kunne prøvekjøre senere, er vi avhengig av klassene *Venn* og *Kontroll*. Derfor begynner vi med dem.

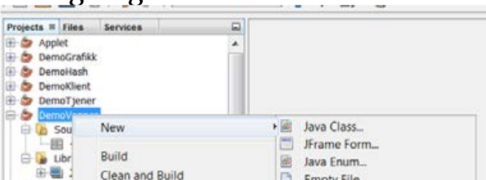
Lag en ny klasse og pass på at den havner i riktig pakke (høyreklikk på pakken):



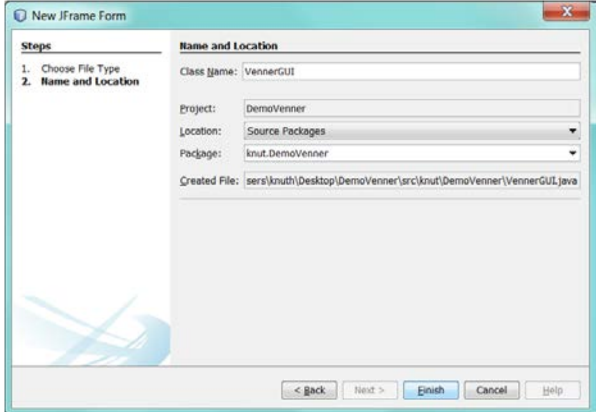
Kall klassene henholdsvis Venn og Kontroll, og skriv koden selv. Bruk NetBeans til å generere kode og rett den opp.

3. Lag et vindu

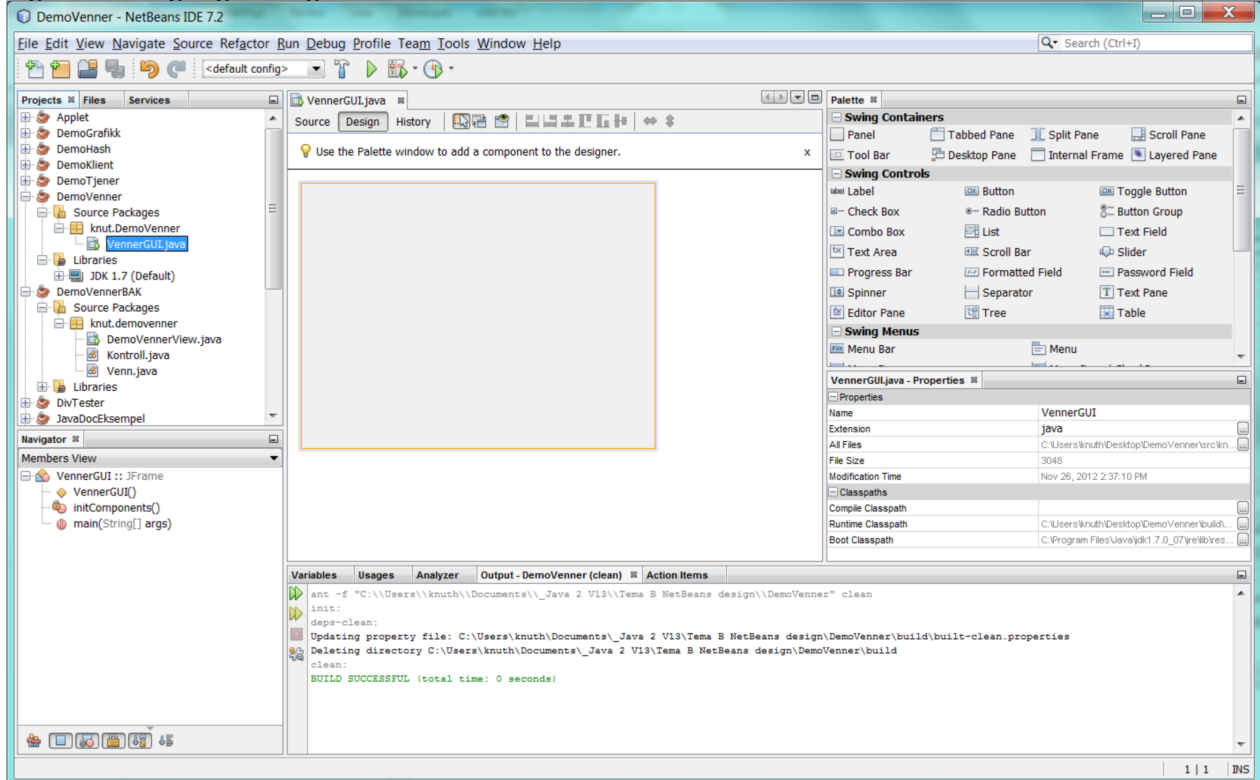
Høyreklikk prosjektet "Venner" og velg en *JFrame Form*:



I veiviseren skal du angi et fornuftig navn for *JFrame*. Ifølge klassediagrammet skal det hete "VennerGUI". Videre *må* du bestemme et pakkenavn (det advares strengt mot *ikke* å gjøre det – da havner alt i <default package> uten pakkenavn og det skaper problemer senere). Da mange lager klasser med samme navn og noen av dem kan havne i ditt prosjekt fra et bibliotek du anvender, er det konvensjon å benytte eget domene i omvendt rekkefølge foran. Jeg skulle da kalle min pakke for "name.hansson.knut.demovenner" men ulempen med det er at det lages en subkatalog "name" med subkatalog "hansson" osv. og det blir ganske håpløst for et lite prosjekt. Jeg har derfor bare skrevet "knut.DemoVenner":



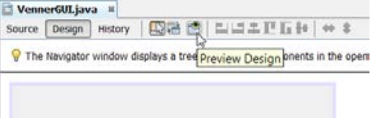
Og da er vi i gang i design mode:



Det er ikke spesielt vanskelig å gi *JFrame* et eget ikon, men vi lar det være i dette kurset[13].

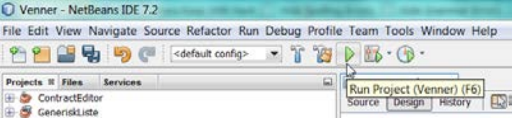
4. Se vinduet og prøvekjør

Du kan se grensesnittet med *Preview Design*. Det er meget raskt og krever ikke kompilering (en fordel når kompilatoren feiler):



[13] Hvis du har lyst til å prøve, så se f.eks. https://stackoverflow.com/questions/1614772/how-to-change-jframe-icon

Du kan også prøvekjøre programmet med knappen, med F6 eller via meny:



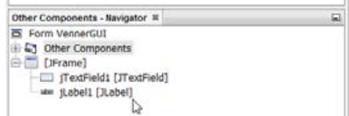
Første gang du prøvekjører må du oppgi *main class* som her er *VennerGUI*. Du kan senere endre dette i prosjektets egenskaper under *Run*.

5. Design skjemaet med Swing-komponenter

Vi kan nå klikke komponenter i vinduet til høyre og plassere dem i skjemaet, sette størrelse osv. NetBeans setter skjemaet til å bruke *GroupLayout* manager som er meget fleksibel, men som er tung å lage selv. Med NetBeans går det imidlertid svært greit.

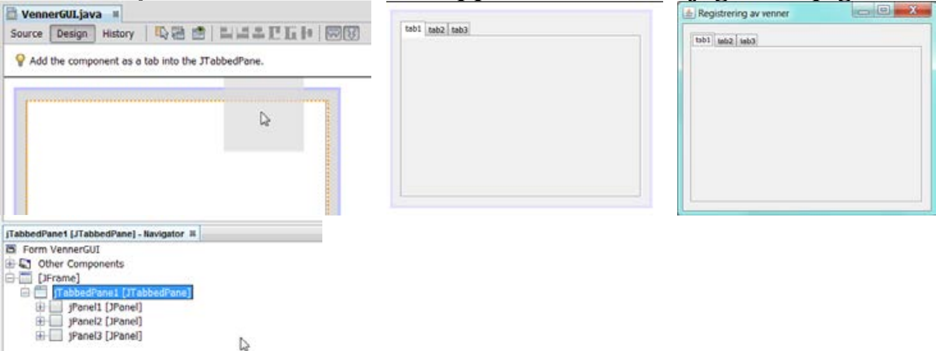
NetBeans har også klargjort skjemaet med CLOSE_ON_EXIT (standard for JFrame er ellers at vinduet skjules men programmet stopper ikke). Det hadde jo vært fikst med en overskrift i skjemaet, så sett egenskapen *title*.

Når du legger til en komponent, inneholder den navnet som tekst. Den vil du jo gjerne fjerne og det bør du gjøre med en gang – komponentene tilpasser seg automatisk størrelsen på innholdet og det kan være vanskelig å rette det opp senere. Skulle noe bli borte for deg når du sletter teksten, så finner du komponentene ned til venstre:



Vi lager en GUI med faner – én for å se alle vennene i en *JTextArea*, én for å endre en venn og én for å slette en venn.

Start med å legge til en *Tabbed pane*. Legg så til fire *Panels*. Når du plasserer dem, så pass på at *Tabbed pane* er merket før du slipper den (her har jeg foreløpig bare tre *Panels*):



Du skal ende med fire *Panels* inne i *Tabbed frame* (bildet helt til høyre viser at jeg har fått det til). Teksten på fanene bør naturligvis endres:

Jeg bryr med (foreløpig) ikke om navnene på komponenter, da jeg ikke skal bruke dem i programkoden.

Nå kan vi legge til komponenter på hver fane. De bør ha "gode" navn. Noen anbefaler at med bruker prefiks (lbl..., txt... osv.) men mange synes ikke det er noe poeng i denne forbindelse. Her vil imidlertid *navn* og *tlf* dukke opp i flere faner, derfor gir jeg prefiks *reg...* på fanen for registrering og videre *slett...*og *endre...*på de andre to.

Jeg tenker å fylle tekstområdet på første fane og komboboksene på de to siste hver gang det skjer en endring – i metoden *oppdater()*.

6. Skriv kode for skjemaet

For å få noen venner å "leke" med, legger jeg til noen venner i *VennerGUIs* konstruktør. Den kalles fra *main*:

```
public VennerGUI() {
    initComponents();

    /*
     * Lagt til av Knut
     */
    try {
        Venn nyVenn;
        nyVenn = new Venn("Knut",22334455);
        Kontroll.kontroll.addVenn(nyVenn);
        nyVenn = new Venn("Arne",11554433);
        Kontroll.kontroll.addVenn(nyVenn);
        nyVenn = new Venn("Åse",-1);
        Kontroll.kontroll.addVenn(nyVenn);
        oppdater();
    }
    catch (Exception e) {
        JOptionPane.showMessageDialog
                (rootPane, "Feil ved oppstart fordi \n"
                + e.getMessage());
    }
    /*
     * Slutt på kode lagt til av Knut
     */
}
```

Videre legger jeg til en metode *oppdater()* som oppdaterer hele skjemaet. Komboboksene settes på nytt, alle felt slettes og tekstfeltet på første fane fylles med venner som hentes med *Kontroll.kontroll.getVenner()*. Denne kalles av konstruktøren til *VennerGUI* og av de tre knappene når de er ferdig med sine endringer.

For å lage hendelsesmetoder for knappene, lønner det seg å sette "ordentlig" navn på knappene først, f.eks. *slett*. Når man dobbeltklikker på knappen, skapes den tomme metoden

```
private void slettActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
}
```

Da er det bare å fylle på med kode (og fjerne //TODO):

```
private void slettActionPerformed(java.awt.event.ActionEvent evt) {
    try {
    Kontroll.kontroll.delVenn(slettNavn.getSelectedItem().toString());
    }
    catch (Exception e) {
        JOptionPane.showMessageDialog
                (rootPane, "Kunne ikke slette denne vennen fordi \n"
                + e.getMessage());
    }
    oppdater();
}
```

Tilsvarende må gjøres for de andre to knappene, og dermed er oppgaven løst.

Ekstra: EasyPMD for NetBeans

NetBeans angir jo mulige feil i form av krøllstreker under feilen og et merke i margen. Det skjer delvis mens man skriver og delvis under kompilering (filene kompileres når de lagres og før kjøring).

PMD

PMD er en fri programvare plugin som utvider denne kontrollen:

1. Den kontrollerer koden mot flere regler, herunder (iflg. *PMDs* nettside på SourceForge):
    a. Possible bugs - empty try/catch/finally/switch statements
    b. Dead code - unused local variables, parameters and private methods
    c. Suboptimal code - wasteful String/StringBuffer usage
    d. Overcomplicated expressions - unnecessary if statements, for loops that could be while loops
    e. Duplicate code - copied/pasted code means copied/pasted bugs

   Også disse mulige feilene (regelbrudd) markers i margen.
2. Man kan lage sine egne regelsett i tillegg eller som erstatning, f.eks. for å implementere virksomhetens standarder (men det er ikke helt enkelt) eller finne ekstra regelsett (gjerne for Eclipse – regelsettene lages likt for alle IDE)

EasyPMD

En utvidelse av dette er *EasyPMD* som også er en fri programvare plugin. Den anvender *PMD* men har i tillegg den fordelen at den

3. Lager en note i vinduet *Action Items* – samme sted som NetBeans setter "TODO-items".

Da blir det hele enda mer oversiktlig.

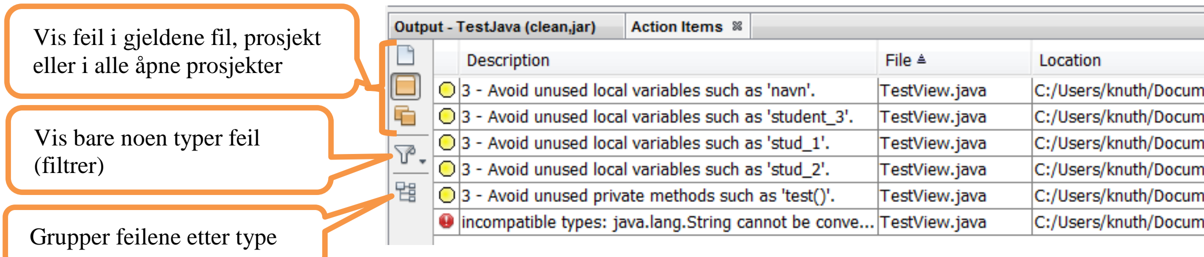
Installasjon og bruk

Du finner informasjon om *EasyPMD* på nettsidene til Gianluca Costa http://gianlucacosta.info/. Sjekk sidene for *My Software|Java*.

Installasjon gjør du direkte i *NetBeans*. På Costas nettsider ser du i en video hvordan det gjøres. Her er en kort oppsummering:

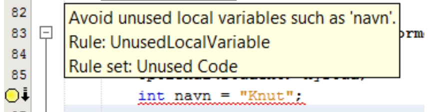
1. Åpne *Tools|Plugins* og fanen for *Available Plugins*
2. Merk *EasyPMD* og klikk *Install*
3. Restart *NetBeans*

Du bruker *EasyPMD* ved å åpne vinduet *Action Items* (i menyen *Windows*). Pass på å gjøre riktige valg i venstre kant av dette vinduet:



Regelbruddene vises uten at du behøver å gjøre noe mer.

I koden vises regelbruddet med et annet symbol enn det NetBeans selv gjør:



Det er også angitt hvilken regel og regelsett som er brutt.

Ekstra: Bruk av komponentbiblioteker i NetBeans

I Swing finnes mange komponenter, men noen ganger kan vi trenge en komponent som Swing ikke har. Det meste lar seg allikevel gjøre, men enkelte ting er tungvinte å få til.

Man undres f.eks. hvorfor Java Swing ikke tilbyr en *datetimepicker*. Man kan jo be brukeren taste inn en dato og så gjøre om det til et *Date*-objekt, men konvertering er svært nøye på formatet som strengen er skrevet inn i, så det blir vanskelig for brukeren. Det blir lettere med ett felt for dag, ett for måned osv. (man ser det ofte gjort slik på nettsider) men det ser ganske primitivt ut.

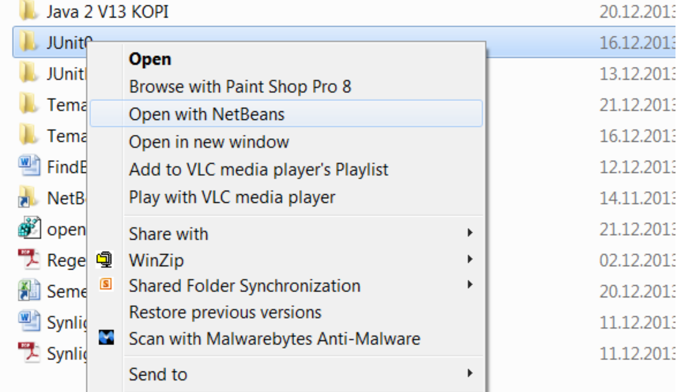
Det kan da være enklere å ty til eksterne biblioteker. Presentasjonen **"Installasjon av biblioteker i NetBeans"** viser hvordan det kan gjøres:

- Flash
- mp4 for iPad
- mp4 for iPhone og iPod

Pass på å kontrollere *lisensen* når du anvender eksterne bibliotek. Det beste er naturligvis *fri programvare* (GNU lisens eller tilsvarende) ellers må du i alle fall være sikker på at du kan distribuere programmet ditt uten å betale ekstra til rettighetshaveren. Hvis du vil distribuere din applikasjon som fri programvare, så *må* også biblioteket være fri programvare – det blir jo en del av ditt program.

Ekstra: Tilpasse Windows Explorer til NetBeans

Det er mulig å få Explorer (utforsker) til å vise "Open with NetBeans" som her:



Da slipper du å åpne NetBeans først for så å finne prosjektet.

Du må endre registeret og det er alltid skummelt, så jeg tar ikke noe ansvar!

Du må lage en ren tekstfil som lagres med etternavn "reg" f.eks. "openwithnetbeans.reg" med følgende innhold:

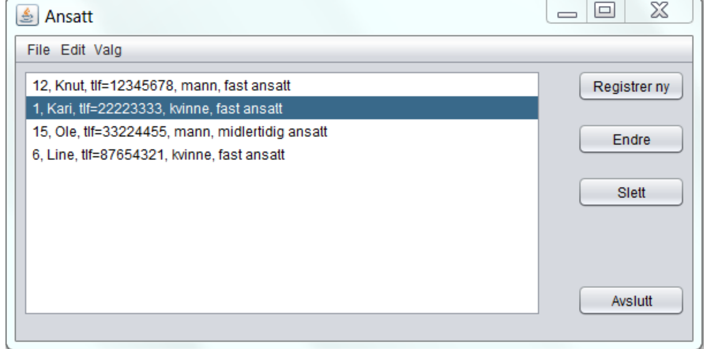
```
Windows Registry Editor Version 5.00

[HKEY_CLASSES_ROOT\*\shell\Open with NetBeans 8.0\command]
@="\"C:\\Program Files\\NetBeans 8.0\\bin\\netbeans64.exe\" --open
\"%1\""

[HKEY_CLASSES_ROOT\Folder\shell\Open with NetBeans 8.0\command]
@="\"C:\\Program Files\\NetBeans 8.0\\bin\\netbeans64.exe\" --open
\"%1\""

; Pass på at sti og programnavn er nøyaktig riktig! Merke doble
backslash!
; For å fjerne tidligere HKEYs så sett et minustegn foran HKEY, f.eks.
; [-HKEY_CLASSES_ROOT\*\shell\Open with NetBeans\command]
; [-HKEY_CLASSES_ROOT\Folder\shell\Open with NetBeans\command]
; (Semikolon innleder kommentar.)
```

Dobbeltklikk på reg-filen i utforsker så legges disse nøklene til i registeret.

Det er den ytterste mappen du må høyreklikke på så NetBeans finner alle filene i prosjektet.

Oppgave til kapittel 2

Det skal lages et system for registrering av ansatte, kalt *OppgNetBeans*. Hovedskjema og programklasse (med *main*) heter *OppgNetBeansView*. Det er en JFrame og ser slik ut:



Skjemaet har en menylinje med tre hovedmenyer:

File
- Avslutt
  - Avslutter programmet

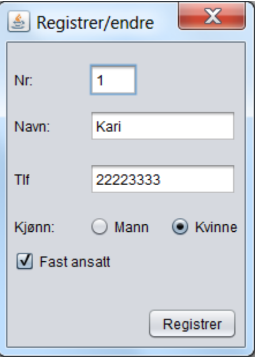
Edit
- Registrer ny
  - Viser et skjema for registrering av ny ansatt med default verdier
- Endre
  - Viser det samme skjemaet som under 2a med eksisterende verdier
- Slett
  - Viser en meldingsboks for bekreftelse

Valg
- Velg farge...
  - Viser en *ColorChooser* for valg av tekstfarge i ansattlisten

Videre er det knapper med de samme handlinger som i menyene unntatt *Velg farge*.

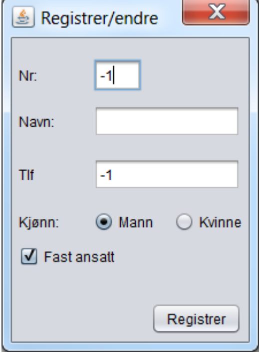
I en *JList* vises alle registrerte ansatte uten spesiell sortering.

Når bruker har markert en ansatt i listen ved å klikke på den (i figuren ovenfor er det nr 1, Kari som er markert) og klikker *Endre* skal den ansatte vises i et dialogvindu *AnsattView*:



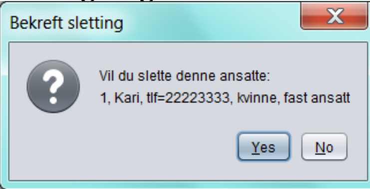
Her kan brukeren endre alle opplysninger og klikke *Registrer*. Da skal dataene bli endret og det vises i hovedskjemaet. Hvis brukeren bare lukker vinduet med "lukkeknappen" skal intet endres.

Hvis brukerne klikker *Registrer ny* skal det samme skjemaet vises, men nå med default-verdier:



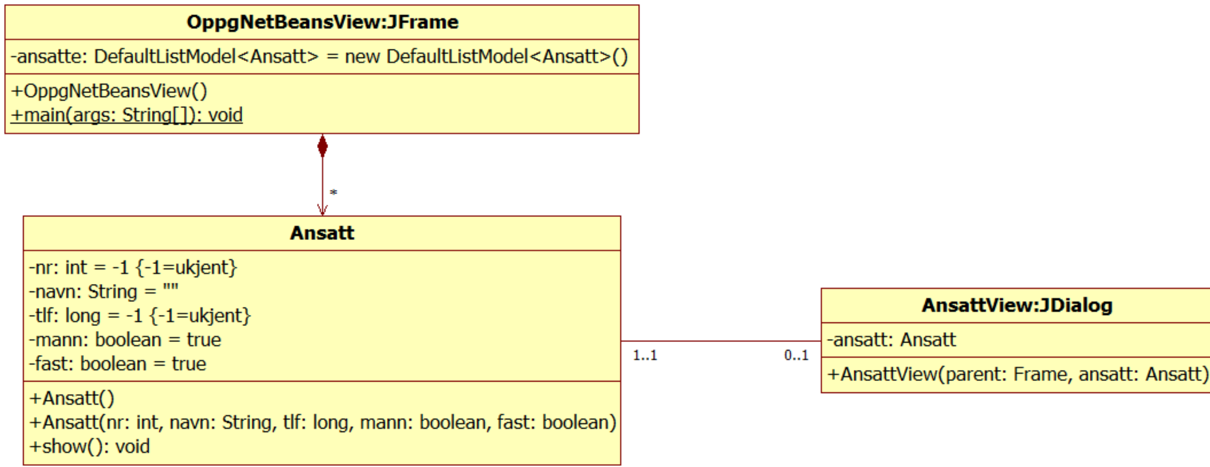
Brukeren fyller ut og klikker *Registrer* for å registrere den nyansatte. Du behøver ikke å kontrollere input (se nedenfor under utfordring). Hvis brukeren bare lukker vinduet med "lukkeknappen" skal intet registreres.

Hvis brukeren har markert en ansatt og velger *Slett* vises en meldingsboks:



Yes gir sletting, *No* gir ikke sletting.

Klassediagrammet ser slik ut:



Siden det bare er én entitetsklasse, vil en egen kontrollklasse bare gi et ekstra ledd uten interessant funksjon. Du lar heller *JFrame* ta seg av å holde orden på de ansatte i en *DeafultListModel*.

Her er ikke alle tilgangsmetodene vist, og *Ansatt* må overstyre *toString()* – du ser i listen ovenfor hvordan strengen skal vises. Det er heller ikke tatt med de forskjellige lytterne med *ActionPerformed*-metodene.

Din oppgave: Lag systemet. Det skal ikke lagres noe.

Vink/råd

For å gjøre det enklere å prøvekjøre, har jeg laget følgende setninger i hovedskjemaets konstruktør:

```
ansatte.addElement(new Ansatt(12,"Knut",12345678,true,true));
ansatte.addElement(new Ansatt(1,"Kari",22223333,false,true));
ansatte.addElement(new Ansatt(15,"Ole",33224455,true,false));
ansatte.addElement(new Ansatt(6,"Line",87654321,false,true));
```

Det er de ansatte du ser i listen i bildet ovenfor.

I hovedskjemaet er det en *JList*. Radene i en slik kontroll er lagret i en *DefaultListModel<E>*. Ved å ha *ansatte* deklarert som en slik modell, kan vi knytte den til listen med *setModel(ansatte)*. Da vil alle endringer i "modellen" *ansatte* reflekteres synlig i kontrollen. Endringer i objektene endres ikke, bare endringer i selve listen. Det innebærer at kontrollen oppdateres når en ansatt legges til (*addElement*), fjernes (*remove*) eller settes (*setElementAt*). Vi slipper da å ha en egen *ArrayList* eller lignende og så fylle kontrollen fra den.

Det er lagt opp til at ansatte har en metode *show()* som viser den ansatte i dialogskjemaet. Den brukes til å vise dialogskjemaet når det klikkes *Endre*. Hvis brukeren klikker *Registrer* i dialogskjemaet, endres den lokale variabelen *ansatt* som er overført som argument til

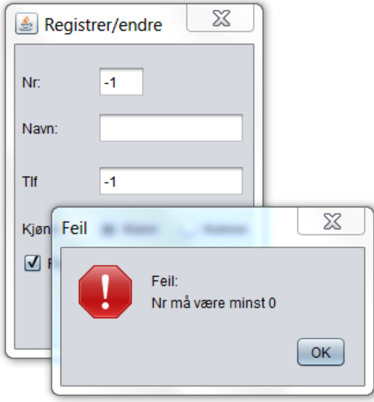
konstruktøren. Når dialogen lukkes returneres det til hovedskjemaet og "modellen" *ansatte* oppdateres.

Hvis det skal registreres en ny, må det først lages en midlertidig ansatt, f.eks. *tmpAnsatt*, med standardverdier (bruk den argumentfrie konstruktøren *Ansatt()* til det). Da kan den kalles med *tmpAnsatt.show()*. Ved retur må den legges til "modellen".

Ved sletting brukes en helt vanlig *ConfirmDialog* som lages med *JOptionPane* og den ansatte fjernes evt. fra "modellen".

Utfordring

Prøv å kontrollere input både ved registrering av nyansatt og endring, slik at nr>0, at navnet er minst ett tegn og ikke *null* og at tlf enten er større enn 0 eller -1. Ved feil, vises passende feilmelding, f.eks.:



Hvis brukeren kansellerer eller lukker vinduet, skal en nyansatt eller endring ikke registreres.

Du kan gjøre om slik at *show()* returnerer true/false. For å få *show()* til å gjøre det, kan du prøve å skape en ansatt med de verdiene brukeren har angitt. Hvis det går greit kan du returnere *true* eller *false*. Det kan se slik ut (i *show*):

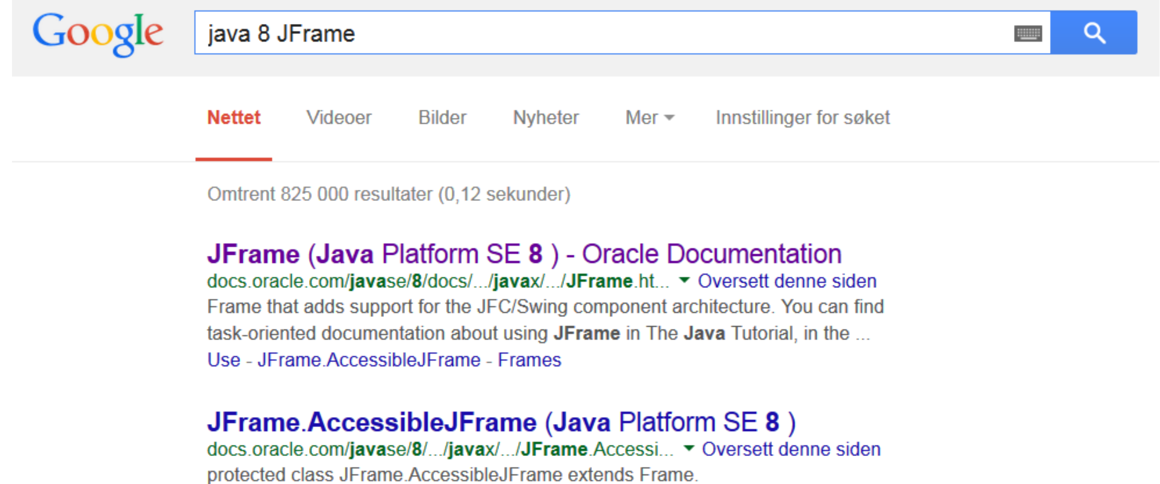
```
try {
   Ansatt temp = new Ansatt
         (getNr(),getNavn(),getTlf(),isMann(),isFast());
   return true; //dette gikk greit
} catch (Exception ex) {
   return false; //klarte ikke å skape en ansatt med disse verdiene
}
```

I løsningsforslaget er det gjort.

Kapittel 3 – JavaDoc

Bruk av Java API Dokumentasjon

Java har et fint, standardisert system for dokumentasjon som er brukt for Java selv. Du finner den komplett på http://docs.oracle.com/javase/8/docs/api/index.html. Jeg synes imidlertid selv at den er litt tung å lete i, og foretrekker å Google f.eks. "java 8 JFrame" og så velge den beste lenken:



Her velger jeg den første som åpenbart er del av Oracles dokumentasjon for Java 8 (der ser jeg av adressen).

Åpne denne siden på deres egen PC.

All slik dokumentasjon er bygget opp likt.

1. Øverst ser vi at *JFrame* er en *klasse* og tilhører biblioteket *javax.swing*. Vi kan se arvehierarkiet (klassen bygger på *java.awt.Frame* som igjen bygger på *java.awt.Window* osv.). Vi ser også hvilke grensesnitt klassen implementerer (*ImageObserver*, *MenuContainer* osv.).
2. Deretter vises deklarasjonen av klassen i sin helhet. Der fremgår de samme opplysningene og i tillegg kan vi se at klassen er konkret (den kan instansieres) ellers ville det stått *public abstract class…*
3. Det følger en omtale av klassen inkludert en referanse til en tutorial (det kan være nyttig) og et par advarsler.
4. Det følger en oversikt over nøstede klasser dvs. indre klasser deklarert innenfor *JFrame*-klassen. Dette er klasser vi kan bruke til å lage objekter av, f.eks. *new JFrame.AccesibleFrame()*. Det må i tilfelle gjøres inne i et *JFrame*-objekt (eller en subklasse), da denne nøstede klassen er *protected*.
5. Deretter følger en oversikt over fields (tilgjengelige egenskaper), konstruktører og metoder. Alle disse er klikkbare for mer detaljer. Vi kan f.eks. se at metoden *setDefaultCloseOperation* returnerer void, krever et heltall som argument og setter hva som skal skje når vinduet lukkes. Default er HIDE_ON_CLOSE. Den kan kaste visse feil og det er forklart når de vil kastes.

Summen av alt dette er en meget nøyaktig og grundig innsikt i klassen. Som regel kan man da finne det man er på jakt etter og løse sitt problem.

Dokumentasjon av grensesnitt

Dokumentasjonen for et grensesnitt er litt annerledes. Vi kan f.eks. se at klassen *ArrayList<E>* som dere har brukt, bl.a. implementerer grensesnittet *Collection<E>*.

Finn selv dokumentasjonen for grensesnittet Collection<E>.

1. Det angis først bibliotek og navn.
2. Vi kan se at *Collection<E>* er et undergrensesnitt av *Iterable<E>* og har mange undergrensesnitt, f.eks. *BeanContent*.
3. Videre ser vi hvilke klasser som implementerer dette grensesnittet, f.eks. altså *ArrayList*.
4. Det følger en omtale av grensesnittet og nyttig informasjon for en som selv vil implementere klassen
5. Metodeoversikten angir hvilke metoder en implementering må definere, og hvordan de skal virke. De er klikkbare for ytterligere informasjon for en som implementerer, inkludert evt. "kontrakter" (se f.eks. *equals*) som spesifiserer nøyaktig hvordan metoden forutsettes å virke.

Det er vanligvis ikke nødvendig å slå opp grensesnitt for en klasse vi skal bruke. Metodene som er hentet fra grensesnittet er jo definert i klassen. Imidlertid sendes vi ofte til grensesnittet hvis vi klikker på en av lenkene i klassen avsnitt om *Methods Inherited from...*

For *ArrayList* finner vi følgende arvede metoder:

Methods inherited from class java.util.AbstractList
equals, hashCode

Methods inherited from class java.util.AbstractCollection
containsAll, toString

Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.List
containsAll, equals, hashCode

Her kan det være spørsmål om hvilken *equals* vi bør sjekke – det er angitt én arvet fra *List* og én arvet fra *AbstractList*. Siden *List* bare er et grensesnitt, er metoden ikke implementert der – det stilles bare krav til den. Derimot i *AbstractList* som er en klasse, må den være definert. Det vil da vanligvis lønne seg å slå opp klassen heller enn grensesnittet (hvis vi har noe valg).

Hvis du klikker på *equals* i *AbstractList* vil du se at *equals* implementerer både *Collection<E>.equals* og *List<E>.equals* (klassen implementerer begge) og at den overstyrer *Object.equals*. Det er angitt nøyaktig hvordan den virker.

Lage egen JavaDoc

JavaDoc er et program som leser ("parser") dine programmer og dokumenterer dem i html-filer. De får fast, standardisert utseende. Dokumentasjonen er en ment for andre programmerere som skal bruke ditt program som ressurs i sitt eget. Det er følgelig først og fremst for klasser som skal gjenbrukes og biblioteker dette er viktig.

Det er bare det som synes utenfor klassen som dokumenteres. Ettersom de fleste attributter deklareres *private* er det ikke mange av dem som kommer med i slik dokumentasjon. Det er derfor først og fremst metoder som dokumenteres, samt noen *public* konstanter og klassen selv. Til sammen utgjør de klassens API – grensesnittet til andre programmer.

JavaDoc finner frem til alt som er *public* og *protected*, inkludert klasser (også indre klasser) og grensesnitt (*interfaces*), konstruktører, metoder og felt (attributter). JavaDoc tar med deklarasjonene og dine dokumentasjonskommentarer (kommentarer skrevet etter en bestemt syntaks). Du kan selv bestemme om du vil dokumentere bare én klasse/grensesnitt

eller hele pakker. JavaDoc legger selv til lenker til dokumentasjonen av klasser og annet som du bruker. Det er mulig, men nok ikke så vanlig, å selv bestemme hvordan dokumentasjonen skal se ut i en såkalt *DocLet*. Jeg anbefaler å bruke standard *DocLet* så andre programmerere kjenner seg igjen, altså ikke lage noe spesielt.

For å legge til brukbar dokumentasjon utover de rene deklarasjonene (som JavaDoc uansett tar med) skriver du kommentarer i et bestemt format.

1. Du skriver kommentarene *rett foran det som skal kommenteres* (ingen tomme linjer mellom din kommentar og deklarasjonen).
2. Kommentarene står mellom /** (merk dobbelt stjerne) og */.
3. Du bruker
   a. vanlig tekst
   b. definerte tagger som begynner med @, f.eks. *@author Knut*. Du kan lage egne tagger i din egen *DocLet* men det er altså ikke vanlig.
   c. HTML-tagger, f.eks. *<p>*, *<img src="bilde.jpg">*, men unngå headinger som *<H1>* o.l. Tegnet < har altså spesiell betydning som starten på en tag i HTML og kan ikke brukes til annet. Hvis du isteden mener at teksten skal inneholde tegnet <, må du bruke litteral-taggen i JavaDoc – se pkt 5 nedenfor.
4. Skriv hver deklarasjon for seg i kildekoden, altså ikke f.eks. *int x, y;* (JavaDoc vil splitte dette i to deklarasjoner som da får samme kommentar.)
5. Tagger kan være
   a. block tag. Den begynner forrest på linjen (evt. etter *), f.eks.
      ** @return int der -1 angir feil*
   b. inline tag. Den står inne i teksten, men må da omsluttes av spissklammer, f.eks.
      ** Hvis a {@literal <}b avsluttes programmet*
6. Hvis JavaDoc ikke finner noe dokumentasjon, vil det lete oppover i arvehierarkiet ditt og implementerte grensesnitt og se om det finner noe passende der.
7. Selve pakken dokumenterer du i en egen fil *package-info.java*.

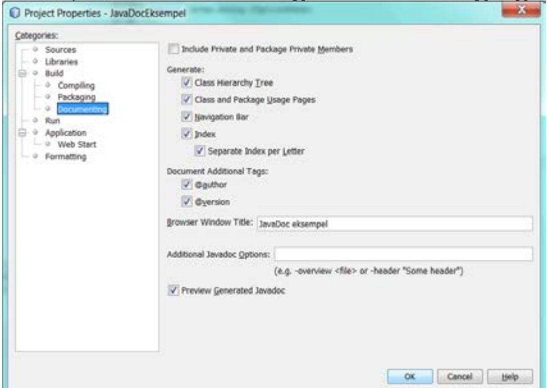
JavaDoc tagger

Se http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html.

De vanligste standardtaggene er (de som står i parenteser er inline tags):

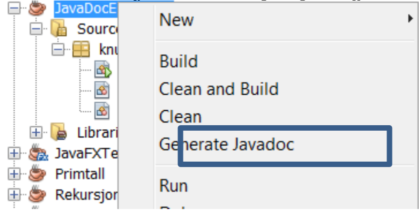
@author	Forfatter
{@code}	Tolk som tekst (ikke f.eks. tag eller html) og bruk programfont.
@deprecated	Vedlikeholdes ikke lenger aktivt (men virker kanskje fortsatt)
@exception	Synonymt med *@throws*.
{@link}	Lenke med etikett til et annet sted i samme dokumentasjonsfil (dvs. vanligvis i samme klasse). Fonten er som programkode.
{@linkplain}	Som *@link* men font som vanlig tekst.
{@literal}	Som *@code* men bruker tekstfont.
@param	Beskrivelse av parametre.
@return	Beskrivelse av returverdier.
@see	Henvisning til annen dokumentasjon med URL.
@throws	Hva slags feil som kastes, med en beskrivelse. Denne legges automatisk til av JavaDoc (forutsatt at metoden er merket *throws*), men da uten beskrivelse.
{@value}	Verdien av en konstant.
@version	Versjonsnummer. Dette kan automatiseres, men dere kan like godt sette det inn selv.

For å få med *@author* og *@version* i NetBeans, bruk menyen *Run|Set Project Configuration|Customize…*, velg *Documenting* og hak ut der:



Generere din JavaDoc

I NetBeans generer du JavaDoc ved å høyreklikke på prosjektet.



Dokumentasjonen finnes igjen i mappen *dist/javadoc* (det er som vanlig *index.html* som er hovedfilen). Du kan sette innstillinger for hvordan JavaDoc skal genereres i *Project Properties* under *Build* og *Documenting*.

Eksempel

Jeg har laget et større eksempel kalt *DemoJavaDoc* som er gjengitt nedenfor.

For å dokumentere selve *pakken*, har jeg laget en egen fil *package-info.java* der jeg har skrevet:

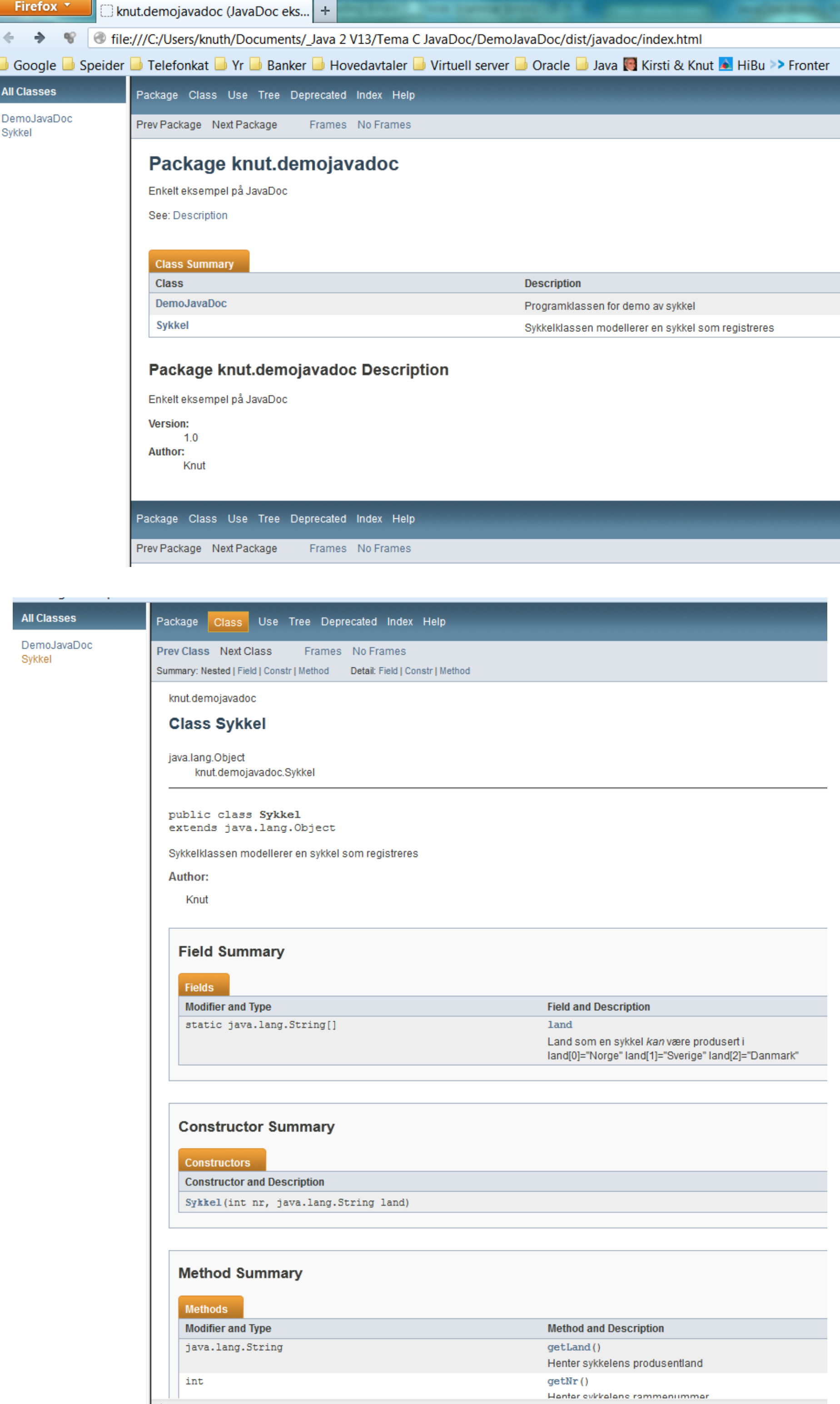
```
/**
 * Enkelt eksempel på JavaDoc
 * @author Knut
 * @version 1.0
 */
package knut.javadoceksempel;
```

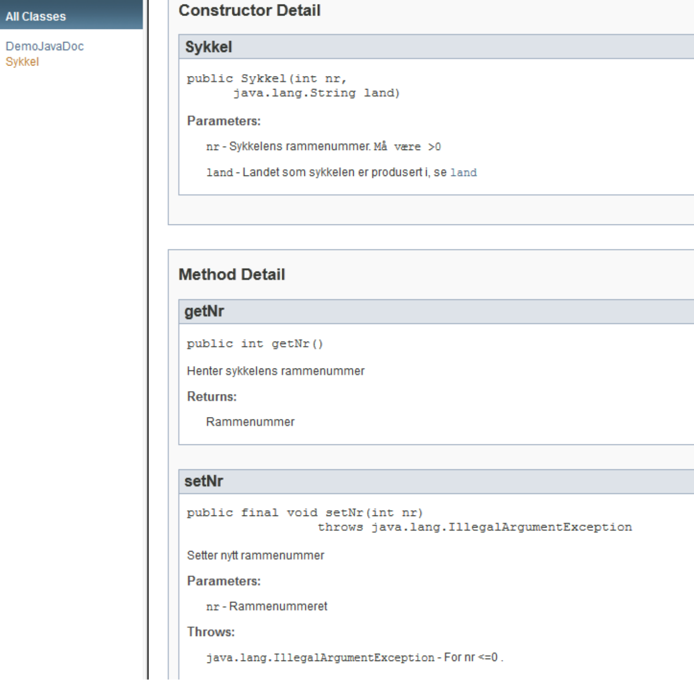
Legg merke til at pakken må deklareres også, ellers genereres ingenting.

Hovedklassen *Sykkel* ser slik ut:

```
package knut.demojavadoc;
/**
 * Sykkelklassen modellerer en sykkel som registreres
 * @author Knut
 */
public class Sykkel {
    /**
     * Land som en sykkel <i>kan</i> være produsert i<br>
     * <ul>
     * <li>land[0]="Norge"</li>
     * <li>land[1]="Sverige"</li>
     * <li>land[2]="Danmark"</li>
     * </ul>
     */
    public static final String[] land = {"Norge", "Sverige", "Danmark"};
    private String produsent;
    private int nr;
    /**
     * @param nr Sykkelens rammenummer. {@code Må være >0}
     * @param land Landet som sykkelen er produsert i, se {@link #land
land}
     */
    public Sykkel(int nr, String land){
        setNr(nr);
        this.produsent = land;
    }
    /**
     * Henter sykkelens rammenummer
     * @return Rammenummer
     */
    public int getNr(){
        return nr;
    }
    /**
     * Setter nytt rammenummer
     * @param nr Rammenummeret
     * @throws IllegalArgumentException For {@literal nr <=0 }.
     */
    public final void setNr(int nr) throws IllegalArgumentException{
        if (nr<=0) {
            throw new IllegalArgumentException
                ("Rammenummeret kan ikke være mindre enn 0");
        }
        this.nr = nr;
    }
    /**
     * Henter sykkelens produsentland
     * @return Produsentland
     */
    public String getLand(){
        return produsent;
    }
    /**
     * @return Sykkelens rammenummer og produksjonsland
     */
    @Override public String toString(){
        return nr + " produsert i " + getLand();
    }
}
```

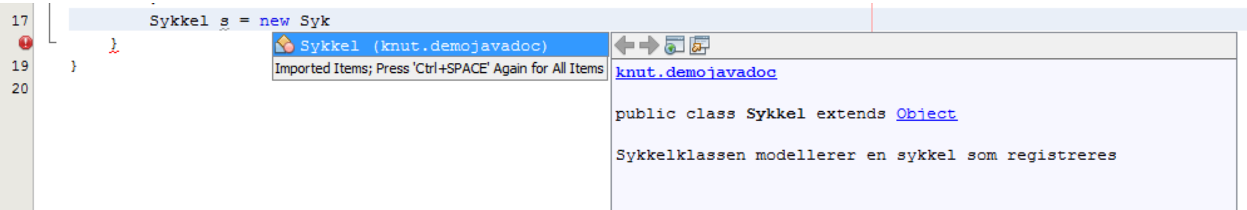
Resultatet vises umiddelbart i nettleseren din (hvis "Preview Generated JavaDoc" er haket ut i *Project Properties*). Her er noen bilder av resultatet:



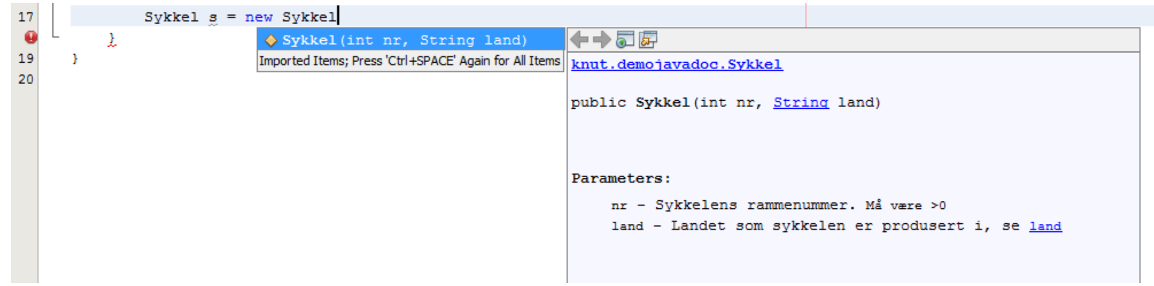


Når dokumentasjonen er laget, får du dokumentasjonen frem under kodingen. Da begynner dette å bli nyttig!

Trinn 1:



Trinn 2:



Trinn 3:

```
15            JOptionPane.showMessageDialog(null, ex.getMessage());
16        }
          Sykkel s = new Sykkel(nr, null)
```
int nr, String land

Ekstra: Likhet og tilordning i Java, argumenter/parametre

Du har tidligere benyttet alt det som er nevnt i overskriften. Her går jeg kanskje litt dypere i stoffet for bedre forståelse.

Likhet

I Java bruker tester man *likhet* på to måter:

Primitive variable (tall, boolean, char)	Objekter	String

`int a = 5;`
`int b = 10;`
`--------------`
`if (a==b) {…`
Sjekker om *verdiene* er like. *a* og *b* må ha samme type eller konverteres.
 	
equals kan ikke brukes på primitiver.
 	
Primitiver kan ikke være *null*.
	`Ansatt a = new Ansatt(5);`
	`Ansatt b = new Ansatt(10);`
	`-------------------------`
	`if (a==b) {…`
	sjekker om a og b har *samme adresse* i RAM, altså om *referansene* er like. Da refererer de til samme objekt (instans).
	
	`if (a.equals(b)) {…`
	Sjekker om objektene er like utfra definisjonen av *equals*. *equals* må overstyres ellers virker den likt med ==. *b* kan være *null* og det skal alltid gi *false*, men ikke *a* (det gir feil).
	
	`if (b==null) {…`
	Sjekker om objektet *b* er *null*.
		`String a = "Per"`
		`String b = "Olga"`
		`--------------------------------`
		Strenger er objekter og likhet oppfører seg som alle andre objekter.
		`if (a==b) {…`
		sjekker om de to strengene har *samme adresse* i RAM. Det er sjelden interessant.
		
		`if (a.equals(b)) {…`
		equals er overstyrt og sjekker om de to strengene er like tegn for tegn. Det er vanligvis det vi ønsker.
		
		`if (b==null) {…`
		Sjekker om objektet, dvs. strengen, *b* er *null*.
		
		`if (b.equals("") {…`
		`if (b.isEmpty()) {…`
		`if (b.length==0) {…`
		Sjekker om strengen finnes men er uten tegn.

Når du overstyrer *equals* bør du passe på at *a.equals(b)* gir true hvis og bare hvis

1. *b* ikke er null
2. *a* og *b* er av samme klasse
3. utvalgte attributter i *a* og *b* har samme verdi

Hvis *hashCode* skal brukes, må du overstyre den også, slik at to objekter som er like, garantert får samme hash-kode. Det vil i prinsippet si at de feltverdiene som avgjør om objektene er like, også benyttes til å generere hash-koden.

Tilordning

Tilordning av **primitiver** gjøres alltid med operatoren =. Hvis en primitiv variabel tilordnes en annen variabel, må de være av samme type (eller konverteres). Da er det verdien av den ene som tilordnes den andre. Etterpå har de samme verdi men deler ikke samme plassering i RAM. Lokale variable som er primitiver initieres ikke implisitt – vi må skrive det selv.

Tilordning av **objekter** kan gjøres med *new* og konstruktøren for objektets klasse. Da skapes et nytt objekt på et nytt, ledig sted i RAM. Objekter kan også tilordnes andre objekter av samme type med operatoren =. Da settes de to referansene til å peke til samme sted i RAM (de blir "alias") og sammenlikningen == blir *true*.

Strenger er "sånn litt primitiv og litt objekt". Det er jo lov å skape med =, mens objekter alltid må skapes/instansieres med *new*. På den annen side gir det underlige resultater:

```
String s1= "abc";
String s2= "abc";
if (s1==s2) {…
```

Her skal if-testen sjekke om de to strengene har samme RAM-adresse. Den gir *true* fordi de to like strengene legges samme sted i RAM når tilordningen gjøres slik. Den første setningen skaper en konstant streng "abc" og *s1* settes til å peke på den. Den andre setningen setter *s2* til å peke til den samme konstanten.

Men...

```
String s1= new String("abc");
String s2= new String("abc");
if (s1==s2) {…
```

Her vil if-testen gi *false* fordi de to strengene skapes på to, forskjellige steder i RAM når konstruktøren brukes.

s1.equals(s2) gir *true* i begge tilfeller.

Parametre/argumenter

Med **parameter** – som ofte kalles *formelt parameter* – mener vi en verdi som en metode forventer å få overført, f.eks. *x* i

```
void doble(int x){
```

Med **argument** – som gjerne kalles *aktuelt argument* – mener vi hvilken verdi metoden faktisk gis som input, f.eks. 5-tallet i setningen:

```
doble(5);
```

eller *a* i setningen:

```
doble(a);
```

Argumentet må naturligvis være av samme type som det formelle parameteret (eller en subtype eller en type som kan konverteres automatisk til parametertypen).

Primitiver som parameter.

Vi har følgende metode

```
void doble(int x){
    x=2*x;
}
```

Den kaller vi slik:

```
int a=5;
doble(a);
```

Etterpå er *a* fortsatt 5. Hvorfor? Det er fordi det skapes en *kopi av a* i RAM før kallet. Det er *adressen til kopien* som overføres til *doble(x)* der variabelen *x* – som er lokal – settes til å peke på *kopien av a*. Når da variabelen dobles, er det *kopien av a* som dobles. Det har ingen effekt på *a*.

Dette kalles *reference by value*. Det er den eneste måten å overføre argumenter på i Java.

Hva kan gjøres? Vi kan jo gjøre om *doble* til en funksjon, slik:

```
int doble(int x){
    return 2*x;
}
```

Den kaller vi slik:

```
int a=5;
a=doble(a);
```

Nå skapes det også en *kopi av a*. Den overføres til *doble* som kaller den *x* og gir tilbake den dobbelte verdien av *x*. Den returnerte verdien tilordnes *a* i siste setning, og *a* er blitt 10.

Objekter som parameter

Anta at vi har en metode som bruker Ansatt som parameter:

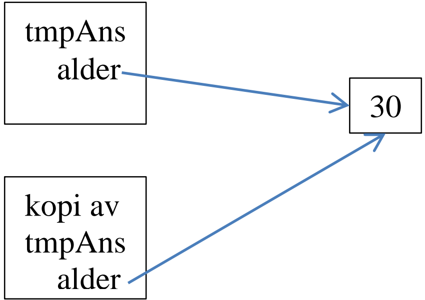
```
void fødselsdag(Ansatt ansatt){
    ansatt.alder++;
}
```

Den kaller vi slik:

```
Ansatt tmpAns = new Ansatt(30); //30 er alderen
fødselsdag(tmpAns);
```

Etter det vi så ovenfor om *reference by value*, skulle man forvente at *tmpAns.alder* fortsatt var 30 etterpå. Den er blitt 31! Hva skjer her?

Jo, det lages fortsatt en kopi av argumentet, dvs. en kopi av ansatt-objektet *temp*. Denne kopien er en såkalt *shallow copy* dvs. at objektet kopieres bit for bit. Men da peker jo variabelen *alder* fortsatt til samme sted. Altså er det den verdien som *fødselsdag* endrer og det er den samme som den originale *tmpAns* peker til.



fødselsdag kaller kopien for *ansatt* og endrer objekts variabel *alder* til 31. Etterpå oppdager vi at også *tmpAns* – som peker til det samme tallet som kopien – faktisk er blitt endret.

Når vi gir objekter som argument, kan altså metoden endre de objektfeltene som den får tilgang til.

Strenger har den egenskapen er at de er *immutable*. Det betyr at de ikke kan endres. Det kan virke lit rart, for vi kan jo skrive f.eks.

```
String tmpAns = "Knut";
tmpAns = "Kari"; //vi har ombestemt oss
```

I linje to får jo *tmpAns* vitterlig ny verdi.

Poenget er at det ikke er slik at det er strengobjektet som *tmpAns* peker på som endres. Det skapes et helt nytt objekt – med verdien "Kari" – og *tmpAns* settes til å referere til dette *nye* objektet.

Hvis objektets variable er en streng, blir det da annerledes enn for objektet ovenfor (med en tallvariabel). Anta f.eks. at den ansatte også har et navn. Vi har følgende metode:

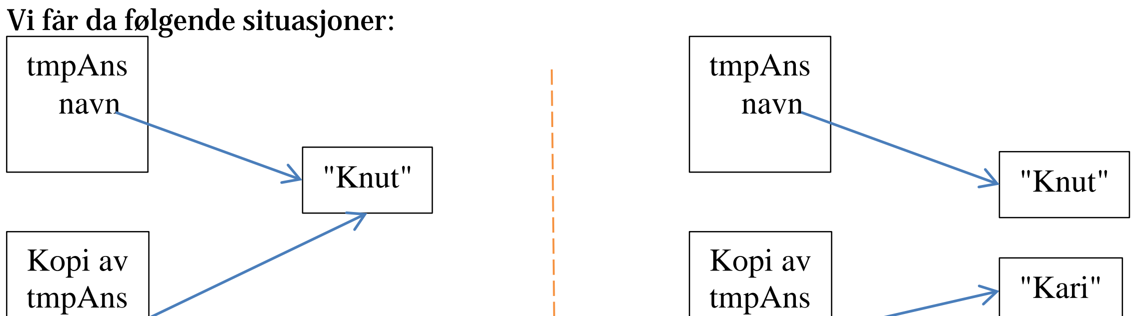
```
void nyttNavn(Ansatt ansatt){
    ansatt.navn="Kari";
}
```

Vi bruker den slik:

```
Ansatt tmpAns = new Ansatt("Knut"); //Knut er den ansattes navn
nyttNavn(tmpAns);
```

Nå vil vi vente – siden argumentet er et objekt og navn er en variabel i objektet, at *tmpAns.navn* er blitt endret etter kallet. Det er det ikke! Hva skjer her?

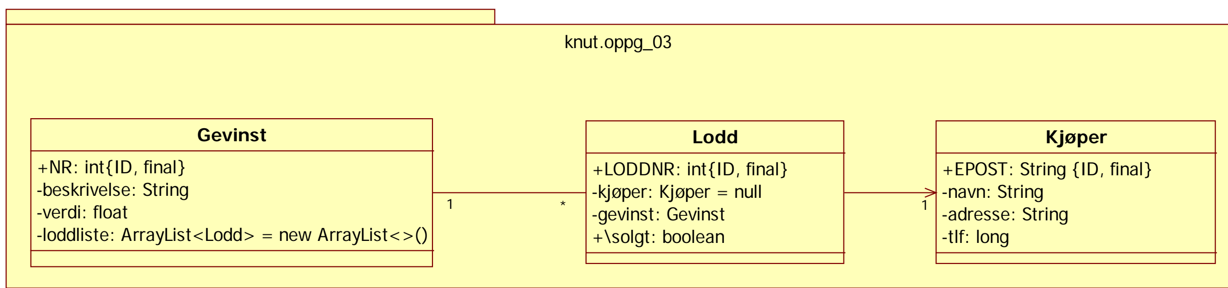
Vi får da følgende situasjoner:



Til venstre er situasjonen *før* kallet. De to peker til samme streng. Til høyre er situasjonen *etter* kallet. *nyttNavn* har skapt en ny streng og refererer til den. Det berører ikke originalen.

Oppgave til kapittel 3

Det skal lages et klassebibliotek for lotterier. Klassediagrammet ser slik ut (det er svært forenklet for at dette ikke skal bli for stor oppgave):



Det er laget ferdig noen stubber nedenfor. Dem kan du kopiere inn i ditt eget NetBeans prosjekt.

Din oppgave

1. Lag et prosjekt i NetBeans. Det enkleste er å skape tomme filer og så lime inn kildekoden som er vedlagt nedenfor. Du vil se at det er laget en metode eller to, men det meste mangler og *isSolgt()* er ikke ferdig.
2. Skap alle *getter*- og *setter*-metodene med NetBeans. Noen av metodene må tilpasses litt og noen skal ikke være med.
3. Skap *toString, hashCode* og *equals* for alle klassene.
4. Sjekk at alt kompilerer greit.
5. Manglende importer ordnes med NetBeans.
6. Skriv fulle kommentarer for alt som er public og generer JavaDoc for biblioteket.
7. Da *Kjøper.NR* er *final* og *public*, trengs det ikke *getNr*. Tilsvarende for Lodd og Gevinst.

Package-info.java

```
package knut.oppgjavadoc;
```

Kjøper.java

```
package knut.oppgjavadoc;
public class Kjøper {
    public final String EPOST;
    private String navn;
    private String adresse;
    private long tlf;
}
```

Lodd.java

```
package knut.oppgjavadoc;
public class Lodd {
    public final int LODDNR;
    private Kjøper kjøper=null;
    private Gevinst gevinst;
    public boolean isSolgt() {
        //TODO: return true/false
    }
}
```

Gevinst.java

```
package knut.oppgjavadoc;
public class Gevinst {
    public final int NR; //ID
    private String beskrivelse;
    private float verdi;
    private ArrayList<Lodd> loddliste = new ArrayList<>();

    public Lodd findLodd(int nr){
        for(Lodd lodd:loddliste){
            if (lodd.LODDNR==nr){
                return lodd;
            }
        }
        return null;
    }
    public ArrayList<Lodd> getLoddliste(){
        return loddliste;
    }
    public void addLodd(Lodd lodd){
        loddliste.add(lodd);
    }
    public void delLodd(Lodd lodd){
        loddliste.remove(lodd);
    }
}
```

Kapittel 4 – Abstrakte datatyper (ADT)

Vi kan samle forskjellige data i ett objekt (nr, navn, adresse…). Det er da ett eksemplar av hvert data i hvert objekt. Det vil vi *ikke* kalle en samling, men objektvariable (*object fields*) som Java kaller dem.

Hvis vi samler *flere objekter av samme type i ett objekt*, f.eks. mange strenger i en String[], kalles den en samling (*collection*). Slike samlinger utgjør i seg selv et objekt, men inneholder altså mange objekter av samme type[14] i en bestemt *struktur*.

Strukturene kan beskrives som forskjellige matematiske modeller som beskriver samlingens egenskaper, de operasjoner som er definert for dem, operasjonenes resultat og noen ganger også "kostnaden" i ressursbruk. Hvordan de faktisk er implementert – eller kan implementeres – sier modellene ingenting om. De er altså *abstrakte*. Slike abstrakte modeller kalles *ADT* (*abstract data types*).

Et eksempel kan være *heltall*. Det kan f.eks. defineres som et tall som er delelig med 1 eller som et tall uten brøk eller desimaltegn. Deretter definerer man de operasjonene som skal gjelde for heltall: addisjon, subtraksjon osv. og hvordan de skal virke. Alt dette gjøres av matematikere helt uten tanke på hvordan det eventuelt skal implementeres i et programmeringsspråk. Når Java-konstruktørene mener at de trenger et slikt heltall, finner de på et navn (*int*) og definerer et utvalg av operasjonene enten som operatorer (+, - osv.) eller funksjoner sqrt().

Her skal jeg først beskrive noen slike ADT. Senere beskriver jeg de grensesnitt (*interface*) og klassene som Java tilbyr for de forskjellige ADT-ene.

ADT Mengde (*set*)

Mengde er en samling objekter uten ordning (rekkefølge) der ingen er like. Objektene i mengden kalles vanligvis elementer. Operasjonene som er definert, er mengdeoperatorene Union, Snitt og Komplement. Man kan spørre om et objekt er element i mengden eller ikke. Det skal være mulig å legge til og fjerne et element. Videre *kan* det være definert en operator som returnerer antall elementer og en operator for differanse. Hvis mengden tillater et element å være *null* (det varierer) kan det bare være én slik (ellers blir det flere like).

ADT Bag

En bag har samme definisjon som en mengde, men kan inneholde like elementer (dubletter). Hvis bagen tillater elementer å være *null* (det varierer) så kan det være flere slike.

ADT Graf

I en graf kalles objektene *noder*. De er knyttet sammen med *kanter*. Kantene kan være retningsbestemt – bare én vei, eller begge veier. Operasjonene som er aktuelle er å legge til/slette en node, knytte sammen to noder og hente antall noder. Det kan også hentes ut en del av grafen – en subgraf. Hver node kan ha egenskaper (attributter) og en peker til en annen node eller flere. Pekeren må ikke ha verdi men det varierer om nodens attributter må ha verdi. Videre kan man følge kantene fra node til node, man "navigerer"/"blar" i grafen.

I implementeringer vil nodene gjerne inneholde et objekt og det varierer om det kan være *null*. De som ikke tillater objektet å være *null* vil alltid være pakket (*dense*), så når en node skal slettes så vil den faktisk fjernes fra grafen – ikke bare nullstilles. Nodens peker

---

[14] I samlinger angir vi *type*. Kompilatoren kontrollerer at vi bare legger objekter av den angitte typen inn i samlingen. Vi kan også legge inn *subtyper* men ikke *metatyper*. Samlinger som er deklarert til å inneholde objekter av typen *Object* kan følgelig inneholde objekter av en hvilken som helst type, siden alle objekter er subtype av *Object*. Det gir stor fleksibilitet, men liten kontroll, så vi vil vanligvis velge lavest mulig type i arvehierarkiet.

implementeres som en referanse (som tillates eller ikke tillates å være *null*) og det det kan være en samling slike referanser.

Siden nodene er lenket sammen etter hvert som de legges til grafen, kan de hentes ut i en rekkefølge som er bestemt av rekkefølgen de ble lagt inn i (på en eller annen måte).

Det finnes mange varianter av grafer. Her er noen av dem:

- ✓ Trær
  Tær har en rotnode som er knyttet sammen med en/flere noder (nodens "barn"), som igjen er knyttet til en/flere osv. i et hierarki. De nodene som ikke har barn, kalles blader. Det er ingen sykler (ingen har sine foreldre, besteforeldre osv. som barn). Hvis hver node har maksimalt to barn, kalles treet et B-tre og hvis alle bladene er (nesten) like langt fra roten kalles treet balansert. Teoretisk sett er mange strukturer varianter av trær:
  - Liste
    I en liste har hver node bare ett barn. Nodene ligger altså på rekke og rad. Roten kalles da ofte "hode".
  - Toveis liste (*deque*).
    Deque uttales som *deck* (kortstokk) og er et akronym for *"double ended queue"*. Listen har da pekere begge veier mellom nodene og har prinsipielt ingen hale, men to hoder. Det gjør det enkelt å legge til/fjerne noder i begge ender.
  - Kø
    En kø er en liste der nye noder legges til i den ene enden og fjernes i den andre. Det virker derfor som en "rettferdig kø" der den som ble lagt til først også fjernes først – FIFO ("first in first out").
  - Stack
    Dette er en liste der den som ble lagt til sist fjernes først – LIFO ("last in first out"). Det er altså en ekstremt "urettferdig kø". Den er praktisk f.eks. når programmet gjør et hopp til et annet sted og vil "huske hvor den kom fra så den kan returnere til der den kom fra sist".
  - Stjerne
    Det er kun ett nivå fra rot til samtlige blader. Det var svært aktuelt den gang vi hadde mini- og stormaskiner direkte tilknyttet et antall terminaler. Stormaskinen sjekket alle terminalene i tur og orden om de hadde noen melding som ventet ("polling").
- ✓ Nettverk
  I nettverk er hver node knyttet til flere andre og det kan være sykler. Internett er typisk et nettverk, det samme er sosiale nettverk. Nettverk har ingen rot, men kan ha blader. En variant av nettverk er
  - Sirkler
    Alle nodene har akkurat ett barn[15]. Da utgjør de en full sirkel og det er hverken noen rot eller noen blad. Det gamle "token ring" lokalnettet sendte rundt en "token" som ga den som hadde den rett til å sende en melding ut på nettet. Melding var adressert til én node men ble mottatt og videresendt av alle.

[15] Dette er en svært forenklet fremstilling av grafteoriens definisjon av sirkel.

ADT figurer

Vanlig tre

Liste

Toveis liste

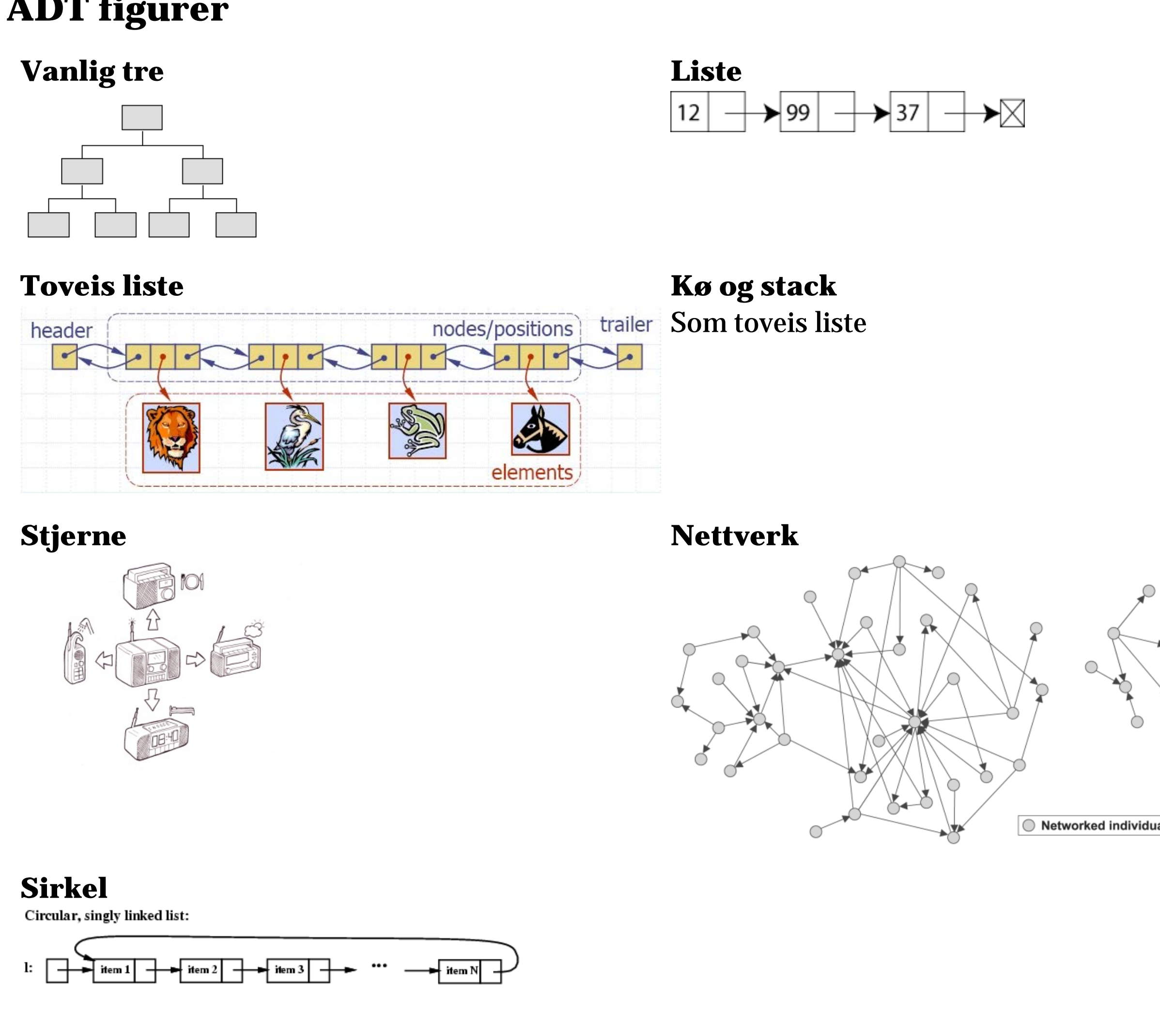
Kø og stack

Som toveis liste

Stjerne

Nettverk

Sirkel



ADT Array

En array har fast størrelse og alle elementene har verdi, evt. null. Den er således ikke "pakket" (den er *sparse*). Man kan verken sette inn eller fjerne elementer, bare "nullstille" dem – vanligvis settes de da til *null* eller en ulovlig verdi, f.eks. alder til -1. Når man går igjennom en array, kan man derfor få elementer som "nullstilt" og som man vanligvis ikke ønsker å behandle. Man kan bare hente/endre et element ved å angi dens plass i arrayen, dvs. dens indeks og det er meget raskt.

Ulempen er at arrayen kan gå full, og da må den utvides – det krever at man skaper en ny og større array og så kopierer alle elementer fra den minste til den nye (og nullstiller resten av elementene i den nye). Det er ressurskrevende. Man kan "pakke" en array ved å kopiere mellom elementer så alle de med verdi ligger etter hverandre forrest, men arrayen inneholder fortsatt like mange elementer (men alle de "nullstilte" ligger bak). Man kan få oppgitt arrayens størrelse, men ikke hvor mange som har verdi eller hvor de ligger.

ADT Hashtabell (*Hash Table*)
I en hashtabell brukes en nøkkel tilknyttet hvert objekt, f.eks. et id-attributt.

Objektene lagres i et antall strukturer med et fast antall plasser, f.eks. arrays, som kalles "bøtter"(*buckets*). Bøttene får hvert sitt nummer fra 0. Nøkkelen brukes som underlag for en hashingalgoritme som angir hvilket bøttenummer objektet skal lagres i. F.eks. kan man dele nøkkelverdien med antallet bøtter og bruke resten som bøttenummer. Hvis f.eks. nøkkelen er 12 og det er fem bøtter, så skal objektet finnes/lagres i bøtte 2 (12 dividert med 5 gir 2 til rest), objekt med nøkkel 15 skal i bøtte 0 osv. Når en bøtte blir full bruker man en egen bøtte for *overflow* med ubegrenset plass – gjerne en liste.

Man prøver å bruke en hashingalgoritme som sprer objektene jevnes mulig til bøttene, for lengst mulig å unngå at en bøtte blir full. Det er tungt å finne objekter som har havnet i overflow. Det har vist seg at hashingalgoritmen (og derved også antallet bøtter i vårt eksempel) bør baseres på et primtall da det gir jevnest fordeling av hash-tallene. I noen få tilfeller, der det er lite antall verdier som skal lagres basert på en verdi som er et unikt heltall (en *int* som er ID), kan man basere hashingalgoritmen på en tallverdi direkte – hver verdi havner da i sin egen bøtte. Denne hashingalgoritmen kalles "triviell".

Kapasitet angir hvor mange objekter det er plass til og fyllingsgrad angir hvor mange objekter man forventer å lagre i forhold til kapasitet. Man angir kapasitet og fyllingsgrad som gir en "passe" sannsynlighet for at bøtter blir overfylte.

Ettersom objektene spres utover i bøtter, får de ingen spesiell rekkefølge når alle hentes ut igjen. Det er i motsetning til grafer, der rekkefølgen objektene legges til i, bestemmer rekkefølgen de hentes ut. Når det er en ulempe, lages det i tillegg en sortert indeks over alle nøkkelverdiene.

ADT Andre strukturer
Det finnes mange andre, spesialiserte ADT-strukturer[16]. Vi kan selvsagt også lage våre egne som passer for bestemte formål. Det er imidlertid vanlig å benytte klasser som Java tilbyr – det er betydelig enklere slik.

Forskjellen på lister og arrays
Det er noen prinsipielle og sentrale forskjeller på arrays og lister. Det gjelder størrelse, tilgang til innholdet og pakking.

Arrays har fast størrelse og setter av plass til et visst antall ved opprettelsen. Da må alle elementene i arrayen ha en verdi (evt. nullstilles avhengig av type), de tar plass og de finnes hele tiden. Vi kan ikke fjerne et element, bare nullstille det. Denne arrayen har plass til fem elementer av typen *int* og alle har verdi:

x(0)	x(1)	x(2)	x(3)	x(4)
15	10	3	7	9

Etter at x(1) og x(3) er "slettet" ved å settes til -1 (som her er en "ulovlig" verdi), har arrayen fortsatt fem elementer med følgende innhold:

x(0)	x(1)	x(2)	x(3)	x(4)
15	-1	3	-1	9

Hverken x(1) eller x(3) er egentlig slettet – programmereren må selv holde orden på om -1 er vanlige data eller skal signalisere at elementet ikke skal regnes med. Man vil benytte en verdi som helt sikker er ulovlig til dette.

[16] Interesserte henvises spesielt til *graph theory* på Internett.

Pakking av arrayen vil her innebære at x(2) flyttes til x(1), x(4) flyttes til x(2) samt at x(3) og x(4) "slettes":

x(0)	x(1)	x(2)	x(3)	x(4)
15	3	9	-1	-1

Programmer får tilgang til et bestemt element ved å oppgi indeksen til elementet, så x(0) har verdien 15. Det er meget raskt da alle elementene har lik størrelse, så det er lett å regne ut hvor element 4 er i RAM – de lagres alltid samlet. Til gjengjeld kan arrayen gå full og man kan angi en indeks som er for stor. Da kan arrayen utvides men det fører til kopiering av alle eksisterende elementer til en ny, større array et annet sted i RAM og det er tungt. Det er også tungt å holde en array sortert, da det innebærer å "lage plass" inne i arrayen til nye elementer eller å legge til på første "ledige" plass og så sortere hele arrayen. Anta f.eks. at arrayen ovenfor var sortert:

x(0)	x(1)	x(2)	x(3)	x(4)
3	9	15	-1	-1

For å få lagt inn tallet 6 sortert, må man først flytte alle elementer som er større enn 6 mot høyre:

x(0)	x(1)	x(2)	x(3)	x(4)
3	9	9	15	-1

og deretter sette x(1) til 6. Man må også passe på at det er plass til den nye verdien.

Jeg har bestemt inntrykk av at den generelle arrayen som f.eks. deklareres slik:

```
private String[] x = new String[15];
```

er lite brukt. Årsaken er at man selv må holde styr på så mye: Hvor mange plasser er i bruk, når går arrayen full og hva vi gjør med det, pakking når et objekt fjernes og diverse annet. *De passer derfor best for samlinger som har et fast antall objekter og endres lite.* Eksempler kan være en liste med navn på ukedager/måneder, en liste over brukere som leses inn bare én gang, eller for overføring av et antall argumenter (jfr. *main(String[] args)* som henter evt. argumenter gitt i kommandolinjen ved programstart). Fordelen er da at de er raske, tar liten ekstra plass og har lite "overhead". Vi skal også se arrayen brukt når vi kommer til testing senere i boken.

Allikevel er arrays interessant, fordi det er flere klasser for ADT som har en (skjult) array i bakgrunnen. Klassen inneholder da (skjult) kode som ordner opp i de forhold som er listet som ulemper ovenfor.

I en **liste** opprettes kun listehodet – nodene legges til etter hvert på forskjellige steder i RAM (de ligger ikke samlet). Hvis en node slettes så fjernes den fra RAM og antallet i listen reduseres. De samme dataene som ovenfor i en liste vil se slik ut (pilene er referanser):

Hode →	15 →	10 →	3 →	7 →	9 →	null

Etter at to elementer (verdiene 10 og 7) er slettet, ser listen slik ut:

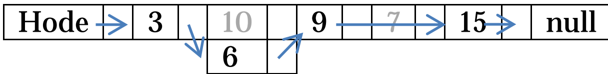
Hode →	15	~~10~~ →	3	~~7~~ →	9 →	null

Element 15 peker nå direkte til element 3, som peker direkte til element 9. Elementene 10 og 7 er det ikke lenger noen referanse til, og de vil bli fristilt av Garbage Collector ved leilighet. Denne listen inneholder nå bare tre noder. Den er altså automatisk pakket.

Videre er det enkelt å legge til et element sortert. Anta f.eks. at ovenstående liste var sortert:

Hode →	3	~~10~~ →	9	~~7~~ →	15 →	null

Vi skal legge til verdien 6 mellom 3 og 9. Det nye elementet settes til å peke på det samme som elementet 3. Deretter settes elementet 3 til å peke på det nye elementet:



I en liste får man kun tilgang til en node ved å starte forrest og bla videre til neste inntil man er fremme ved riktig node. (I toveislister kan man bla begge veier.) For store lister er det tidkrevende. Til gjengjeld går en liste aldri full.

I **hybrider** forsøker man å oppnå fordelene ved begge, ved at listen i virkeligheten er realisert med en array "bak kulissene". Den får man ikke tak i, men man kan oppgi en indeks og få tilgang til en node direkte. Hybrider holder selv orden på hvilke elementer som er slettet, pakker arrayen ved behov og utvider den om nødvendig. Det er fortsatt mulig å oppgi for stor indeks hvilket vil være en feil (*Exception*).

Oversikt over Java-grensesnitt for samlinger

Det er 14 *grensesnitt* for samlinger i Java. De fleste bygger på grensesnittet *Collection*. Det finnes også grensesnitt for hash-strukturer som Java kaller *Map*. Det vil føre for langt å gå inn på alle disse grensesnittene her og vi bruker dem lite, så vi skal bare merke oss at

- ✓ Noen samlinger kan endres (legg til/fjern) og kalles *modifiable*, noen kan ikke endres og er *unmodifiable*.
- ✓ Samlinger som i tillegg aldri endres synlig er *immutable*, de andre er *mutable*.
- ✓ Lister som alltid har samme størrelse selv om elementene endres, er *fixed-size* de andre er *variable-sized*.
- ✓ Lister som gir rask tilgang etter indeks, kalles *random access lists*, de andre er *sequential access lists*.

Man bør også merke seg at grensesnittet *Set* ikke tilsvarer mengder, da de mangler operatorene union, snitt og komplement. Slik sett er *Set* en skuffelse. *Set* er bare en liten utvidelse av det helt generelle *Collection* ved at *Set* ikke tillater dubletter.

Noen grensesnitt er begrenset til visse typer av objekter, andre kan kreve at objektene ikke er *null* eller at de oppfyller andre betingelser.

Hvis vi skal anvende et slikt grensesnitt, må vi naturligvis lage vår egen klasse som implementerer grensesnittet (eller flere av dem).

Oversikt over Java-klasser for samlinger

Av større interesse for oss er de 10 ferdige *klassene* som kan instansieres direkte til vår bruk. Følgende tabell, hentet fra dokumentasjonen, gir en oversikt:

Grensesnitt	**Implementert klasse**				
	Hashtabell	**Variable-sized array**	**Balansert tre**	**Liste**	**Kombinert hashtabell og liste**
1. List		***ArrayList***		***LinkedList***	
2. Map	***HashMap***		***TreeMap***		LinkedHashMap
3. Set	HashSet		***TreeSet***		LinkedHashSet
4. Deque		***ArrayDeque***		LinkedList	

Det er ingen begrensning i type objekter i noen av dem – man oppgir hvilken type objekter de skal inneholde når man instansierer dem (med *new*).

Man har dessuten fortsatt to "gamle" strukturer som er "pusset opp": *Vector* og *HashTable*. Mitt inntrykk er at de "er gått av moten", men mange bruker dem sikkert ennå fordi de er godt kjent med dem og de kan bli brukt mer nå som de er "pusset opp".

I denne boken skal vi etter hvert se litt på seks av klassene (med fet kursiv i tabellen) samt Vector:

1. Lister
   A. LinkedList
   B. ArrayList
   C. Vector
2. Map
   A. HashMap
   B. TreeMap
3. Set
   A. TreeSet
4. Deque
   A. ArrayDeque

Iterator

Iterator er et grensesnitt som alle de ovennevnte samlingene støtter. Dere har sett på dette grensesnittet i et underliggende kurs. Den har metoder som *hasNext*, *next* og *remove*. De er nyttige hvis man vil bla igjennom en samlings verdier eller nøkler. F.eks. har dere sikkert brukt Iterator slik (studenter er en *ArrayList*):

```
Iterator<Student> minIterator = studenter.iterator();
while (minIterator.hasNext()){
     System.out.println(minIterator.next().toString());
}
```

Note: Personlig ville jeg heller brukt en "foreach-struktur" her[17] fordi jeg synes den er enklere å lese/forstå. Senere (kapittel 13) viser jeg enda ett alternativ, nemlig en strøm kombinert med Lambda-uttrykk. Da oppnår du andre fordeler i tillegg til enklere kode.

Oppgave til kapittel 4

Vi skal lage en stakk helt fra bunnen av (uten å bruke de ferdige samlingene). Den abstrakte datatypen *Stakk* er en *liste* med *noder* der man legger til nye elementer og fjerner dem fra samme ende. Stakk har operatorene *push* som legger til en node og *pop* som henter en node (og fjerner den fra stakken). Vi skal realisere denne ADTen.

Vi lager en klasse *Stakk* med metodene *push(objekt)* som legger til et objekt øverst i stakken og *pop()* som henter objektet øverst og fjerner det fra stakken. Vår Stakk skal også ha *peep()* som henter øverste objekt uten å fjerne det fra stakken, samt *getStakk()* som returnerer alle objekter i stakken som en *Vector*. Alt skal typebestemmes generisk.

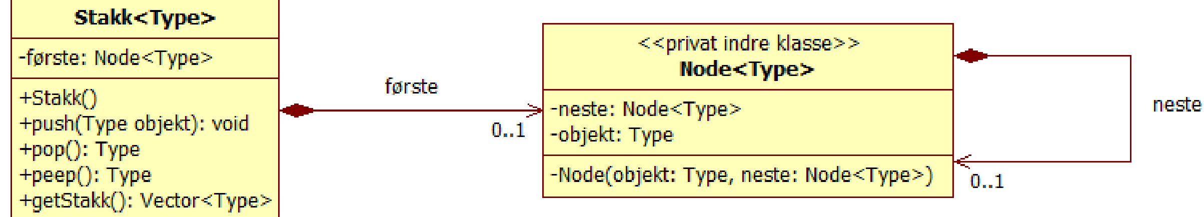
Beskrivelse og klassediagram

Til nodene brukes en privat, indre klasse med plass til objektet og en referanse til "neste" node.

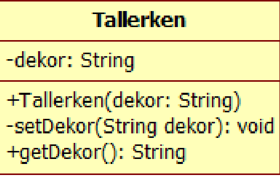
Objekter og returverdien tillates å være *null*. Hvis stakkens "første" er *null* så er stakken tom, og hvis en nodes "neste" er *null* så er denne noden den siste i listen.

[17]
```
for (Student student: studenter){
    System.out.println(student);
}
```

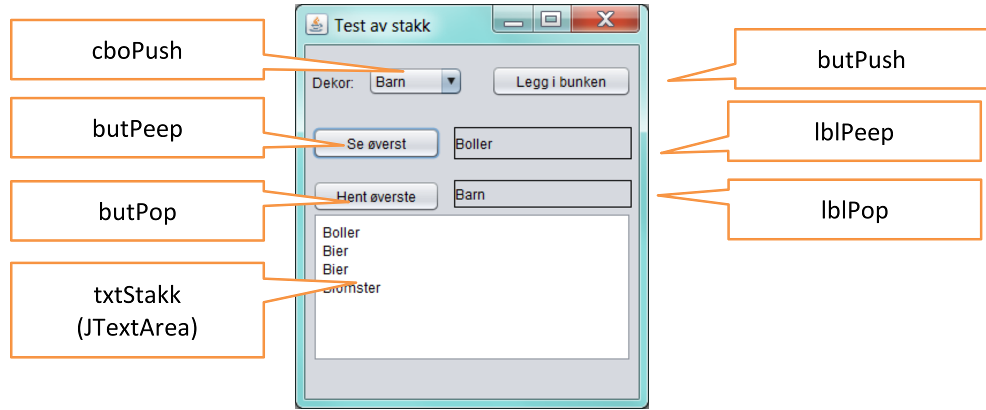
Klassen *Stakk* blir da slik, med *Node* som indre klasse:



Når den er klar, prøver vi ut denne stakken med en bunke tallerkener (da er *Type* = *Tallerken*). Tallerkenklassen ser slik ut:



Grensesnittet for testing er en *JFrame* som heter *OppgStakkView* og ser slik ut:



Brukeren kan velge i komboboksen mellom forskjellig dekor, f.eks. Blomster, Bier, Barn osv. (som er strenger) og legge en ny tallerken i bunken. Videre kan han se på øverste som vises i *lblPeep*, og hente/fjerne øverste som vises i *lblPop*. I tekstboksen *txtStakk* vises hele tiden stakken etter at operasjonen er gjennomført.

Lag en kontrollklasse som holder orden på stakken og la grensesnittklassen kalle den. Når du har laget den ferdig, vil du nok se at en slik kontrollklasse er "overkill" her. Det vil bare føre til mye sending av data mellom grensesnittet og kontrollklassen uten at vi oppnår vesentlige fordeler. Den eneste fordelen jeg kan se, er at det lett kan legges til nye grensesnitt, f.eks. for mobile enheter, uten at kontroll-logikken behøver å endres eller kopieres. Når det bare skal lages ett grensesnitt og det bare er én entitetsklasse, blir gjerne kontrollklassen overflødig. Kontroll-logikken kan da like gjerne inkluderes i grensesnittklassen.

Kapittel 5 – Lister inkl. Vector i Java

I dette kapitlet ser vi bare på punkt 1 *lister* – de andre tas opp i neste kapittel.

Metodene som de enkelte klassene tilbyr, overlater jeg til studentene å slå opp. I tabellen i kapittel 4 (side 49) – og under hver klasse nedenfor – er det klikkbare lenker til dokumentasjonen av hver klasse.

Det finnes flere typer lister i Java – hvilken bør man bruke?

En "ekte" enveisliste bør ha referanse til den første noden (*first* = null) og kunne skape noder med plass til (a) et objekt og (b) en referanse til neste node (evt. null). Det er egentlig alt vi trenger. En slik liste har ubegrenset kapasitet og pakker nodene. I praksis har ferdige listeklasser ofte mange flere metoder som gjør programmeringen enklere.

De faktiske, konkrete listeklassene er

A. LinkedList
B. ArrayList
C. Vector

De har ikke like egenskaper og passer til forskjellig formål.

Ad A: LinkedList

Dette er tilsynelatende en "ekte" liste. Den har ubegrenset kapasitet, pakker nodene, gjør det mulig å finne første (og siste) og å bla fra node til node. Man kan legge til ny node forrest og bakerst.

LinkedList har mange metoder som gjør programmeringen enklere. Bl.a. tillates man å oppgi indeks, men dokumentasjonen forklarer at man da vil traversere listen korteste vei, enten forfra eller bakfra[18]. Det vil altså koste eksekveringstid i store lister.

Listen er strengt typet, men alle typer kan legges inn. F.eks.

```
LinkedList<Student> studenter = new LinkedList<>();
```
[19]

Man advares spesielt om at *LinkedList* ikke er synkronisert. Det innebærer at hvis flere tråder aksesserer listen, og den ene av dem endrer den strukturelt ved å legge til eller slette en node, så vil den andre tråden kaste feil. (Dette kalles "fail fast" i motsetning til "fail safe"). Man kan trygt endre verdier i nodene. Dokumentasjonen forklarer hvordan man kan sikre synkronisering når listen skapes, i fall man skal benytte flere tråder.

LinkedList kan også benyttes som en kø og som stack. Dette kalles en "deque" ("double ended queue") uttales *dekk*. Derfor har klassen metoder som "kikker på" første/siste node uten å fjerne den, og som henter og fjerner første/siste element. Ved å legge til nye i den ene enden og hente fra den andre, bruker man den som en kø. Hvis man legger til ny og henter i den samme enden, er det i praksis en stack.

For detaljer om metoder, se dokumentasjonen på
http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

Ad B: ArrayList

ArrayList er en hybrid – det er en skjult (*private*) array bak listen. Det har fordeler ved henting med indeks, da programmet går direkte inn i arrayen og finner noden. Imidlertid har arrayen selvsagt fast kapasitet slik at den kan gå full. Da vil *ArrayList* selv utvide arrayen

---

[18] Det tyder på at man faktisk har realisert LinkedList som en ekte liste og ikke som en hybrid.
[19] Merk at fra versjon JDK7 er det redundant å oppgi typen flere ganger – derfor "empty caret" etter new.

(dokumentasjonen vil ikke si hvor mye, men andre steder antydes 50 % økning[20]) og det tar tid. Man anbefales derfor å sette kapasiteten tilstrekkelig stor før man legger inn mange noder. Det tilbys metoder for det. På den måten kan man unngå for mange utvidelser.

Videre angis det at arrayen alltid vokser og er alltid minst så stor som den største indeksen som har vært i bruk. Arrayen pakkes altså ikke, men *ArrayList* fremstår allikevel som pakket, så programmet må tydeligvis holde orden på hvilke elementer i arrayen som representerer slettede noder i listen. Metoden *size* angir antallet *noder* (ikke arrayens størrelse). Det er allikevel mulig å redusere arrayen så den passer til antallet noder i listen med *trimToSize*. Det er selvsagt ressurskrevende.

ArrayList er også usynkronisert slik som *LinkedList*.

For detaljer se dokumentasjonen på
http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Ad C: Vector

Vector er (nå – før var den en ren array) en hybrid som likner svært på *ArrayList*. Den har imidlertid bedre muligheter til å kontrollere hvor mye arrayen skal vokse når den går full, og den er synkronisert. I noen sammenhenger – når det skal brukes tråder som oppdaterer samme data – er det en stor fordel, men det kan også oppnås på andre måter (med *wrapper*-klasser).

Full dokumentasjon på http://docs.oracle.com/javase/8/docs/api/java/util/Vector.html

Valgets kval

I valget mellom *Vector* og *ArrayList* sier dokumentasjonen at man bør velge *ArrayList* om ikke synkronisering er et problem. Det vil si at i våre tilfelle vil den nesten alltid være å foretrekke.

Valget står altså vanligvis mellom *LinkedList* og *ArrayList*. Å legge til elementer (noder) inne i en *ArrayList* krever mye overhead, da andre elementer må skyves ut for å gi plass. Da vinner *LinkedList*. På den annen side tar det tid å finne et element som man kjenner posisjonen på inne i en *LinkedList*, mens det er mye raskere i en *ArrayList*. Og endelig vil *ArrayList* kreve overhead når kapasiteten må økes og den krever alltid minst så mye plass som nodene krever – ofte betydelig mer. Å sette inn et element bakerst tar omtrent like lang tid, men å sette inn elementet forrest krever mye for en *ArrayList*.

Valget avgjøres følgelig av bruken – velg ArrayList hvis du har mye henting etter indeks og stort sett LinkedList ellers. LinkedList passer best til køer og stacks og når man setter elementer inn inne i listen[21]. Du kan se en grundig drøfting av dette på
http://stackoverflow.com/questions/322715/when-to-use-linkedlist-over-arraylist.

For øvrig savner jeg en liste som automatisk holder listen sortert. C# har *OrderedList* til dette bruk. Java bruker *PriorityQueue* tilsvarende, men det er altså en kø og ikke en liste, og den har veldig få metoder så den er ikke så brukbar. Alternativet er å sette nye noder inn på rett plass, sortert, men det vil være relativt tungt både i *LinkedList* og *ArrayList*. Antakelig er det bedre å sortere ved behov.

---

[20] Kildekoden er int newCapacity = (oldCapacity * 3)/2 + 1; dvs. temmelig nøyaktig 50 %, men dokumentasjonen garanterer altså ikke at dette blir beholdt senere.

[21] Personlig er jeg glad i *LinkedList* fordi jeg synes den likner mest på slik jeg selv i sin tid programmerte det fra bunnen av. Jeg synes også at jeg har mer kontroll på hva som tar tid – det skjer ikke plutselig noe uventet og ukontrollert overhead. Men jeg innser jo at det er helt irrasjonelt☺.

Alle samlinger inkludert lister kan sorteres med klassemetoden

```
Collections.sort(Collection c);
```

Denne bruker "naturlig sortering" dvs. enten tallstørrelsen, Unicode eller *compareTo*.

Hvis du er interessert i å holde noe sortert til enhver tid, så er det bedre å bruke *TreeSet* (se nedenfor).

Noen bra referanser for dem som vil vite mer
http://java.dzone.com/articles/arraylist-vs-linkedlist-vs
http://java.dzone.com/articles/gaplist-%E2%80%93-lightning-fast-list
http://java.dzone.com/articles/hashmap-vs-treemap-vs

Oppgave til kapittel 5

Det er ganske vanlig å ta arkitektur- og designbeslutninger basert på testing. Noen ganger vil man teste at systemet ikke "kneler" med påtrykk fra mange klienter[22], andre ganger er kapasitet eller responstid i fokus. Man lager seg da en testbenk der deler av systemet utsettes for mange transaksjoner – såkalt stresstesting – og noterer tider, gjennomsnittstid mellom feil o.a.

Denne gang tenker vi oss at vi skal velge mellom *ArrayList* og *LinkedList*. Derfor lager vi et program som sjekker hastigheten. Samtidig får vi "lekt oss" litt med klassen *Random* og tilfeldig sortering.

Vi lager en testbenk som genererer f.eks. 10 000 objekter og lagrer dem i den ene strukturen i tilfeldig rekkefølge (det er antakelig raskere å legge dem inn sekvensielt). Deretter søker vi etter hver forekomst i tilfeldig rekkefølge. Til slutt sletter vi alle postene i tilfeldig rekkefølge. Disse tre operasjonene tar vi tiden på og beregner gjennomsnittstiden for hver.

Ved å gjøre det samme for begge strukturene, kan vi sammenlikne hastigheten på dem.

Objektet vi lagrer er brytere (idrettsfolk) og ser slik ut:

Bryter
+nr: int +navn: String = "Abcdef" +vekt: double +mann: boolean
+equals(Object o): booelan +hashCode(): int +compareTo(Object o): int{-1,0,1} +toString(): String

Vi bryr oss ikke om å skjule attributtene denne gang. Bruk NetBeans til å generere metodene. To objekter oppfattes som like hvis de har samme nummer. Vekten er normalfordelt rundt 85 og ca. 40 % er kvinner. Se nedenfor om hvordan du får til det.

Skap de to objektene *ArrayList* og *LinkedList* uten argumenter. Når objektene skal legges inn i tilfeldig rekkefølge, må først alle skapes f.eks. i en array, sorteres tilfeldig og deretter legges inn én for én. Ta tiden fra den første blir *lagt inn* (så ikke sorteringen osv. påvirker tiden). Tilsvarende bør du omsortere arrayen før du søker og igjen før du sletter.

---

[22] Et klassisk eksempel på hva som ellers kan skje, var forsøk med elektronisk valg i Oslo der serveren ga opp da valget begynte for alvor. Systemet var ikke stresstestet før forsøket og tålte ikke påtrykket fra mange klienter. Eksamenskontoret for grunnskolen i Oslo har opplevd akkurat det samme – elevene mistet kontakt med tjeneren mens de holdt på å svare på oppgavene under eksamen (man delte ut papiroppgaver isteden og måtte gi ekstra tid).

Du kan lage én knapp som gjør alle seks operasjonene, eller én knapp for hver. Uansett skal du legge tidene i en *JTextArea*. Noter resultatene fra en kjøring og ta med til neste forelesning, så vi kan sammenlikne med hverandre. Kontrollklasse er unødvendig – bruk grensesnittet.

Bruk av Random

Klassen Random – som du skaper et objekt av, f.eks. *rnd* – har mange metoder som genererer pseudotilfeldige tall. Her passer det å bruke *rnd.nextGaussian()* til å generere normalfordelt vekt for hver bryter. I så fall kan gjennomsnittsvekten $\bar{x}$ være 85 og standardavviket s være 12. Random gir normalfordelte tall med μ=0 og σ=1, men ved å gange med det standardavviket du ønsker og så legge til det snittet du ønsker, får du den fordelingen du er ute etter, f.eks.

```
rnd.nextGaussian()*s + x̄; //normalfordel rundt x̄ med std s
```

Du kan også bruke *nextInt(n)* for å få et pseudotilfeldig heltall fra 0 og oppover til *n-1* (alle er like sannsynlige). Det passer til å angi bryterens kjønn, så det blir f.eks. 40 % kvinner (*mann=false* for fire av ti).

Sortere en array i tilfeldig rekkefølge ("stokke" elementene)

Det finnes flere, gode algoritmer for dette[23]. Dere kan f.eks. bruke Durstenfelds algoritme (kalt "Algorithm P" av Donald Knuth – sjekk om du forstår hva som foregår):

> To shuffle an array *a* of *n* elements (indices 0..n-1):
> **for** *i* **from** $n - 1$ **downto** 1 **do**
> $j \leftarrow$ random integer with $0 \le j \le i$
>   exchange *a*[*j*] and *a*[*i*]

Her passer det å finne en pseudotilfeldig *j* med *nextInt*.

Ta tiden

Du noterer et tidspunkt med nanosekunders nøyaktighet med koden

```
long start=System.nanoTime();
```

Deler du nanosekunder med tusen får du mikrosekunder, med en million får du millisekunder og med en milliard får du sekunder (bruk *double* for resultatet). Velg en som passer – det er i alle fall tungt å lese tall med mange siffer.

[23] Se f.eks. http://en.wikipedia.org/wiki/Fisher-Yates_shuffle

Kapittel 6 – Maps, set og deque
I forrige kapittel viste jeg noen klasser for lister samt klassen Vector. Her skal jeg se på de tre andre typene.

Maps
Java Maps er ikke *Collections*, men hashede strukturer. Man lagrer altså både en nøkkel – som er utgangspunkt for hashingen til bøttene – og et objekt som er verdien man vil lagre. Maps er et grensesnitt, implementert bl.a. i klassene

A. *HashMap*
B. *TreeMap.*

Uansett implementering, så lagrer *Maps* en nøkkel (*Key*) og en verdi (*Value*) i par (*Entry*). Når Map'en skapes må man følgelig angi to klasser, én for nøkkelen og én for verdiene. Man kan hente dataene som en samling nøkler (*keySet*), en samling verdier (*valueSet*) og en samling verdipar med både nøkkel og verdi (*entrySet*). Det er ikke tillatt med duplikatnøkler.

Det er nøkkelverdien som brukes for å bestemme hvor verdien skal lagres. Hvis nøkkelverdien f.eks. er en Integer, så vil først Integers *hashCode*-funksjon brukes på heltallet[24]. Det produserer et heltall (int). Dette tallet brukes som argument til en egen, komplisert hash-funksjon som returnerer et heltall. Dette angir da bøttenummeret. Den kompliserte hash-funksjonen er laget slik at verdiene blir mest mulig jevnt spedt i bøttene. Hvis nøkkelverdien er en egendefinert klasse, bør man altså definere *hashCode* for den, ellers brukes den arvede fra Object[25] eller annet.

Man må ikke endre verdi på nøklene som er i bruk. Java vil ikke garantere hva som skjer om man bryter med dette. Tryggest er det å bruke en klasse som er *immutable* til nøklene, men det er ikke så lett å finne. Man må heller velge fornuftig nøkkel og selv passe på at den ikke endres. Det beste er å hente brukernes ID-attributt fra de objektene som skal lagres, f.eks. ansattnummer for en ansatt. Da kan man jo selv passe på at verdien ikke kan endres (f.eks. ved at set-metoden gjøres *private* eller at feltet gjøres *final* så verdien bare kan settes av konstruktøren).

Man legger til en ny *entry* med *put(K key, V value)*. Den vil erstatte en evt. eksisterende entry med samme nøkkel som den som settes inn, derfor returnerer den verdien som lå der fra før (for kontroll – *null* betyr at det ikke lå noen der med denne nøkkelen). Man henter verdien ved å oppgi nøkkelen *get(Object key)*. Det innebærer at hvis man ikke kjenner nøkkelverdien, så må man hente hele verdisettet og selv lete gjennom det.

Ad A: HashMap
HashMap er den tradisjonelle hashtabellen med "bøtter" der poster med samme hashverdi lagres sammen. Når den skapes kan man klare seg med å angi bare typene for nøklene og for verdiene:

```
private HashMap<Integer, Student> studenter = new HashMap<>();
```

Her tenker jeg å bruke studentnummeret (en *int*) som nøkkel. Legg merke til at jeg ikke kan angi *<int, Student>* fordi *int* ikke er en klasse men en primitiv datatype.

I en HashMap er alle nøklene unike. Legges enn ny inn med samme nøkkel, vil den overskrive eksiterende.

---

[24] *Integers* funksjon for *hashCode* er svært enkel – den returnerer simpelthen tallverdien. Derved er jo hver tallverdi garantert en unik *hashCode.*

[25] Hvordan Objects *hashCode* er laget vil variere fra operativsystem til operativsystem – det programmeres i "Java Native Interface" dvs. tilpasset programmeringsspråket for den aktuelle OS-plattformen – med C, C++, assembler eller noe annet. Vanligvis gjøres det ved å konvertere objektets RAM-adresse til et heltall (int).

Det vil være bedre å angi kapasitet og fyllingsgrad (*load factor*). Kapasiteten angir hvor mange entries det skal være plass til i utgangspunktet. Fyllingsgraden angir hvor fullt det kan bli før hashtabellen omstruktureres. Da blir hele strukturen hashet på nytt og antallet bøtter omtrent dobles.

```
private HashMap<Integer, Student> studenter = new HashMap<>(1000, 0.75f);
```

Her setter jeg av plass til 1000 studenter med nøkler. Når ca. 750 er lagt inn, vil hashtabellen omstruktureres. (Legg spesielt merke til bokstaven *f* bak tallet 0.75. Det angir at tallet er av type *float* som konstruktøren krever. Alternativt kan man "caste" med *(Float)0.73.*)

For å sette inn en student kan jeg f.eks. skrive (*nyStudent* har verdi og *getNr* henter studentnummeret):

```
studenter.put(nyStudent.getNr(),nyStudent);
```

Vi vet lite om rekkefølgen av entries som returneres og rekkefølgen kan endre seg under bruk (ikke minst på grunn av omhashing). Vi må altså se på *HashMap* som usortert.

Full dokumentasjon på http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

Ad B: TreeMap

En Java *TreeMap* oppfører seg som en hashtabell, men dataene lagres i en trestruktur istedenfor bøtter. Det er derfor uaktuelt å angi kapasitet og fyllingsgrad. I motsetning til *HashMap* så er *TreeMap* alltid sortert etter nøkkelen. Det betyr at hvis man bruker et objekt som nøkkel, så må den ha *CompareTo* definert – i egne klasser må vi definere den selv.

Dataene lagres i et såkalt *red-black tree*[26]. Det har den fordel at det er tilnærmet balansert. For et bestemt tre tar det da tilnærmet samme tid å finne et hvilket som helst objekt.

TreeMap har noen flere metoder enn *HashMap* og brukes på samme måte.

Full dokumentasjon på http://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

HashMap eller TreeMap?

HashMap tillater nulls både i nøkkel og verdi. *TreeMap* tillater det ikke (da ville det ikke være mulig å sortere dem). Jeg kan ikke helt forstå når det skulle være behov for det heller. Begge har kun unike nøkler.

Sorteringen av *TreeMap* gjør det noe langsommere å sette inn en ny entry, så man må vurdere om det er verd det. Begge gir meget rask tilgang og begge sikrer mot duplisering av nøkkelen.

TreeMap er sortert etter nøkkelen, mens *hashMap* er usortert.

Konklusjonen er at hvis du vil ha dataene sortert så passer *TreeMap*. Hvis du bare er ute etter rask tilgang, så velg *HashMap*.

Set

Grensesnittet *Set* gir en uordnet samling elementer uten duplikater. Man legger til/fjerner elementer og får feil om elementet finnes fra før.

Samlingen kan konverteres til *array* og annet så man kan bla igjennom den. Det er ingen implementering av mengdeoperatorene union og snitt.

[26] Interesserte kan lese Wikipedia om slike trær på http://en.wikipedia.org/wiki/Red-black_tree

Som med nøklene i *Map* må man ikke endre verdiene til elementer som er lagt til. Her skal vi bare se på klassen *TreeSet* som implementerer *Set*.

Ad TreeSet

TreeSet lagrer objektene i et tilnærmet balansert tre og treet holdes sortert. Derfor må både *equals* og *compareTo*[27] defineres for objektene som skal legges inn.

Innsetting tar litt ekstra tid pga sorteringen, men trestrukturen gir meget rask gjenfinning.

Også *TreeSet* har unike verdier (som *TreeMap*) men i *TreeSet* brukes verdiene selv som nøkkel.

Det finnes mange metoder for konvertering, finne subtrær osv. Se full dokumentasjon på http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html. Vi skal senere (i et eksempel, side 65) vise hvordan vi kan gå igjennom ("traversere") trær rekursivt.

Deque

En deque (uttales som "deck" i "deck of cards" = kortstokk) er en liste som kun kan aksesseres fra hver ende. Det er altså i prinsippet en dobbeltlenket liste – en "double ended queue", derav navnet. Dokumentasjonen sier at den egner seg for køer og stakker. Det finnes to varianter, men her skal jeg bare omtale *ArrayDeque*.

Ad ArrayDeque

Denne har kun metoder for å manipulere første element og siste element (sette inn, hente, og se på). Hvis den brukes som stakk vil man hele tiden bare operere i den ene enden (forrest eller bakerst) og hvis den skal brukes som kø, vil man legges til i den ene enden og hente fra den andre. Det er ikke tillatt å legge *null* inn i listen.

Fordelen med *ArrayDeque* er at den er rask. Dokumentasjonen sier det slik:

> *This class is likely to be faster than Stack when used as a stack, and faster than LinkedList when used as a queue.*

Full dokumentasjon på
http://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html

Ekstra: Performance for de forskjellige samlingstypene

K. V. Ahuja har gjort en benchmark av de forskjellige samlingene i 2008[28]. Her er noen resultater for innsetting og gjennomgang. Det ble ikke testet gjenfinningstid.

Gjennomsnittstider i mikrosekunder (milliondels sekunder):

Type	Sette inn 100 000 unike elementer	Iterere igjennom 100 000 unike elementer
ArrayList	26	8
LinkedList	55	8
TreeSet	126	10
Vector	20	4
HashMap	43	29
TreeMap	99	98

Jeg er overrasket over at det er så mye raskere å sette inn i en ArrayList enn en LinkedList, for det stemmer dårlig med praktiske erfaringer som rapporteres på nett. Det er naturlig at

[27] Se fotnote på side 19 for en enkel måte å lage CompareTo.
[28] http://scrtchpad.files.wordpress.com/2008/10/java-collections-performance-evaluation.pdf

TreeSet er treg ved innsetting da den holdes sortert. Legg merke til hvor godt *Vector* kommer ut. Den kan sågar ha blitt enda bedre i ny versjon. *Maps* er tregere ved iterasjon – de skal da også hente to verdier.

Det er synd at ikke gjenfinning av tilfeldige poster ble målt, men uansett ser vi at man skal opp i meget store datamengder før forskjellene blir merkbare. Alle målingene gir jo bare brøkdeler av millisekunder for 100 000 poster. Man velger altså i praksis struktur etter behov, ikke etter hastighet.

Ekstra: Litt enkel kode for maps

Note: Dette eksemplet er forenklet maksimalt for å gjøre koden enkel. F.eks. er det ingen kontroller og feltene er ikke innkapslet uten tilgangsmetoder. Du må ikke skrive så enkel kode!

Man vil registrere biler i en HashMap. Bilene har bare to felt:

```
public class Bil {
    Registreringsnummer regnr //ID;
    int regår;
}
```

Feltet *regnr* brukes som identifikator. Det er i tillegg laget konstruktør og *toString* er overstyrt.

Det er en egendefinert klasse for registreringsnummeret med to felt som er identifiserende sammen:

```
public class Registreringsnummer {
    String bokstaver; //ID
    int siffer; //ID
}
```

I tillegg har begge klassene definert konstruktør og overstyrt *toString*.

I kontrollklassen (en singleton) lagres bilene i en HashMap og det er laget noen metoder:

```
public class Kontroll {
    public static final Kontroll kontroll = new Kontroll();
    private HashMap<Registreringsnummer,Bil> biler = new HashMap<>();
    private Kontroll() {
    }
    public void addBil(Bil nybil){
        biler.put(nybil.regnr, nybil); //erstatter evt. eksisterende
    }
    public Bil getBil(Registreringsnummer regnr){
        return biler.get(regnr); //evt. null
    }
    public Collection<Bil> getAlle(){
        return biler.values();
    }
}
```

Her brukes HashMaps metoder *put* og *get*.

Når et objekt av den egendefinerte klassen Registreringsnummer skal brukes som nøkkel, må equals og hashcode være definert. Equals sammenlikner de to feltene bokstaver og siffer og er enkel. Hashcode kan se slik ut:

```
@Override
public int hashCode() {
    int hash = 5;
    hash = 47 * hash + bokstaver.hashCode();
    hash = 47 * hash + this.siffer;
    return hash;
}
```

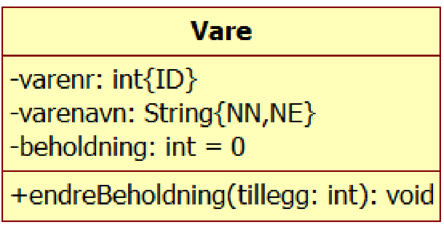
Legg merke til at det er brukt to primtall i algoritmen for å spre verdiene så jevnt som mulig i tillegg til at strengen *bokstavers hashCode* også anvender primtall. Vår *HashCode* returnerer som vanlig et tall. Det er dette tallet HashMap bruker for å finne hvilken "bøtte" bilen skal legges i.

Læreren har komplett kode for dette eksemplet, så det kan vises og prøvekjøres.

Oppgave til kapittel 6

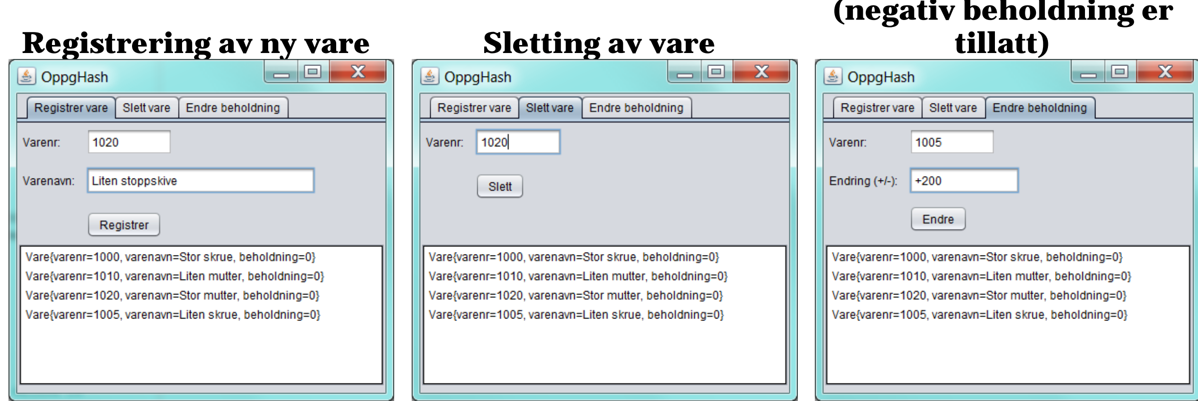
Note: Det blir ikke gitt noen oppgave i Set eller Deque, da løsningen blir svært lik andre som allerede er løst. Her brukes HashMap.

Det skal lages et system for registrering, sletting og oppdatering av varer. Det skal ikke lagres noe til fil/database, men varene skal lagres i en *HashMap*. Klassen *Vare* er dokumentert slik:



I tillegg til den angitte metoden, skal alle tilgangsmetoder lages, samt andre standard metoder.

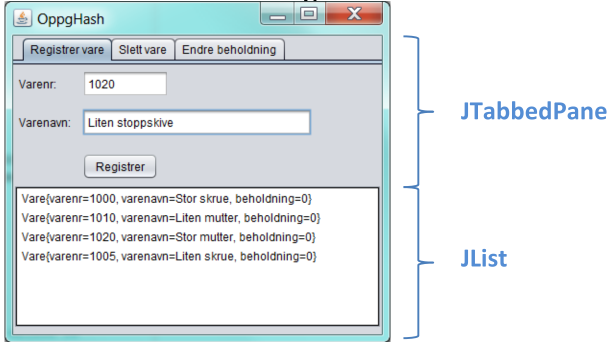
Grensesnittet, som også er programklassen og brukes som kontrollklasse, skal ha faner og se slik ut:



All input skal kontrolleres og feil skal gi feilmelding i meldingsboks. Forsøk på registrering av et varenummer som allerede finnes, forsøk på å slette et varenummer som ikke finnes samt forsøk på å endre en vare som ikke finnes, skal avvises med feilmelding.

Listen over registrerte varer, som synes i alle alternativene, oppdateres kontinuerlig etter alle endinger. Hvis brukeren velger en vare i listen ved å klikke på den, fylles varenummeret ut i fanen for sletting og for endring (men naturligvis *ikke* for registrering av ny vare).

Vink: Grensesnittet laget slik at i den øverste delen vises en *JTabbedPane* med tre faner. Under denne vises en *JList* med alle varene. Derved synes den samme listen uansett hvilken fane som er valgt, slik:



Når det klikkes i en *JList* skjer hendelsen *valueChanged* for listen. Den skriver du kode for og der vil *getSelectedValue()* returnere det objektet som ble merket (det første hvis det er merket flere, null hvis ingen er merket).

Også denne gang skiller du grensesnittet fra kontrollklassen. Se på koden du har laget i kontrollklassen – du vil igjen se at kontrollklassen fremstår som temmelig unødvendig gitt at det ikke skal lages flere grensesnitt eller lages en klient-tjener applikasjon. I kontroll-klassen skriver jeg f.eks.

```
public void put(int varenr,Vare vare){
     varer.put(varenr, vare);
}
```

I grensesnittet kaller jeg denne med

```
Kontroll.kontroll.put(regVare.getVarenr(), regVare);
```

Hvis HashMap varer var deklarert i grensesnittet, ville jeg isteden ha skrevet

```
varer.put(regVare.getVarenr(), regVare);
```

Kontrollobjektet fremstår da bare som et unødvendig ekstraledd.

Kapittel 7 – Rekursjon

Ekstra: Litt repetisjon om funksjoner (for sikkerhets skyld)

En funksjon er en regel som til hvert element i en mengde A (verdimengden) tilordner ett og bare ett element i en mengde B (løsningsmengden). Poenget for en programmerer er altså at funksjonen bruker en oppgitt verdi til å finne én verdi som returneres (funksjonsverdien). Regelen som anvendes er den algoritmen som er skrevet i funksjonen. F.eks.

```
private long dobbelt(int a){
    long verdi = 2 * a;
    return verdi;
}
```

Funksjonen *dobbelt* krever et argument[29] av typen *int* og returnerer en verdi av typen *long*. Returverdien beregnes på grunnlag av den oppgitte verdien som lokalt kalles *a*.

Det er ingenting i veien for at funksjonen kan kreve flere argumenter, gjerne av forskjellig type, men den kan bare returnere én verdi (evt. ett objekt).

```
private long rest(long a, int b){
    return a % b;
}
```

Dere husker kanskje også at "algoritmeteoremet" gjorde det klart at enhver algoritme kan skrives med noen elementære setninger og de tre strukturene sekvens, seleksjon (valg) og iterasjon (gjentakelser, løkker). Det er imidlertid ikke helt sant, for et annet teorem fastslår at enhver iterasjon kan erstattes av en rekursjon og omvendt.

I programmeringsundervisning verden over, har lærere en egen forkjærlighet for rekursjon. Det er litt uklart for meg hvorfor rekursjon skal være så viktig, men jeg innrømmer at jeg synes det er *morsomt*. Det gir også fin algoritmetrening, og noen ganger er det nyttig. Vi modellerer jo egentlig løsninger med våre algoritmer og noen ganger (svært sjelden, riktignok) er det lettere å se for seg modellen rekursivt. F.eks. hender det den følger direkte av en definisjon. Et eksempel er mappestrukturen på harddiskene våre som er slik:

Formelt	Oversatt
Stasjon ← Rotmappe Rotmappe = Mappe Mappe ← 0{Fil | Mappe}n	*Alle stasjoner har en rotmappe.* *En rotmappe er en mappe.* *En mappe inneholder fil(er) og mappe(r).*

Den siste setningen er rekursiv fordi definisjonen av "mappe" inneholder "mappe".

Rekursive funksjoner

En *rekursiv* funksjon er en funksjon der regelen har flere ledd, la oss kalle dem delregler. Forskjellige vilkår avgjør hvilken delregel som skal anvendes. Det må være minst én enkel regel og minst en rekursiv regel. Her er et eksempel (Fibonaccitallene):

1. $Fib_1 = 0$ dvs. "det første tallet i rekken er 0"
2. $Fib_2 = 1$ dvs "det andre tallet i rekken er 1"
3. $Fib_n = Fib_{n-2} + Fib_{n-1}$ for n>2 dvs "alle andre tall i rekken er summen av de to foregående tallene i rekken"

Delregel 1 og 2 er enkle (primitive) regler, mens den tredje er rekursiv fordi *Fib* inngår på begges sider av tilordningen. En matematiker kan finne på å skrive det slik:

$$Fib(n) = \begin{cases} n=1: 0 \\ n=2: 1 \\ n>2: Fib(n-2)+Fib(n-1) \end{cases}$$

Programmeringen av funksjonen *Fib* er da et spørsmål om å følge reglene til punkt og prikke:

[29] Forskjellen på parameter og argument er omtalt på side 37 i kapittel 3.

```
private int fib(int n){
    if (n==1) return 0; //delregel 1 - enkel
    if (n==2) return 1; //delregel 2 - enkel
    return fib(n-2)+fib(n-1); //delregel 3 - rekursjon
}
```

Vi slipper med andre ord å vurdere hvordan dette kan omgjøres til en iterasjon (men det er fullt mulig). Slik vil det eventuelt se ut:

```
private int fib2(int n) { //iterativt
        int pre1=0, pre2=1;
        for(int i=0; i<n-1; i++) {
            int tmpPre = pre1;
            pre1 = pre2;
            pre2 = tmpPre + pre2;
        }
        return pre1;
}
```

Her "simulerer" vi egentlig rekursjonen og denne løsningen synes i alle fall jeg er vanskeligere å komme på.

Demo

Jeg har laget en demo som viser rekursjon i mange sammenhenger.

Eksempel: Potens rekursivt

Potens er vanligvis definert slik at $x^y = x*x*x...$ i alt y ganger. Det kan også defineres rekursivt for y>= 0 som

1. Hvis y=0: $x^0=1$
2. Hvis y>0: $x^y=x*x^{y-1}$

Dette innebærer at f.eks. $3^2=3*3^1=3*3*3^0=3*3*1=9$

```
private int potensrekke(int x, int y){
    //forutsetter y>=0
    if (y==0){
        return 1;
    }
    return potens = x * potensrekke (x, y-1);
}
```

Eksempel: Finn største i en liste

Den største i en liste kan defineres slik (jeg ser bort fra tomme lister):

1. Hvis listen har bare ett element: Elementet er størst
2. Ellers så er den største én av følgende to: Første element i listen eller den største i resten av listen

```
private String finnStørste(List<String> liste){
        //listen forutsettes å innhold minst ett element
        if (liste.size()==1){
            return liste.get(0);
        }
        String størst_i_resten =
                finnStørste(liste.subList(1, liste.size()));
        if (liste.get(0).compareTo(størst_i_resten) > 0) {
            return liste.get(0);
        }
        else {
            return størst_i_resten;
        }
```

```
    //alternativt:
    //return (liste.get(0).compareTo(størst_i_resten) > 0 ?
    //        liste.get(0) : størst_i_resten);
    }
```

Note: I kapittel 13 kommer jeg tilbake til hvordan man kan finne største element i en stor array ved å fordele arbeidet på flere prosessorer med parallelle strømmer (side 117) og med fork/join (side 122).

Eksempel: Multiplikasjon

Multiplikasjon kan beskrives som summering slik for *gange(x,y)* som skal gange x med y og y forutsettes positiv:

1. y=0: gange(x,0)=0
2. y>0: gange(x,y)=x + gange(x, y-1)

```
private int gange (int x, int y){
    //Fortusetter at y er minst 0
    if (y==0){
        return 0;
    }
    return gange (x, y-1) + x;
}
```

Eksempel: Omgjøring fra desimaltall til binært tall

Der lærte vi å dele desimaltallet med 2 og ta vare på resten, deretter dele svaret med 2 og ta vare på resten osv. til svaret ble 0. Svaret skulle leses "baklengs". F.eks.

```
13 : 2 = 6, rest 1
 6 : 2 = 3, rest 0
 3 : 2 = 1, rest 1
 1 : 2 = 0, rest 1
```

Svaret leses oppover og er 1101.

Det hadde vært kjekt å få det første sifferet ut først og ikke sist som her, og det kan vi oppnå ved rekursjon i funksjonen *binært(desimal)* som skal gi en streng

1. desimal=0: ""
2. desimal>0: binært(desimal/2) + desimal%2 som streng

Uttrykket *desimal%2* gir resten når vi dividerer med 2.

```
private String binært(int desimal){
    if (desimal==0){
        return "";
    }
    return binært(desimal / 2) + Integer.toString(desimal % 2);
}
```

Eksempel: "Traversere" trær

Trær har vi snakket om tidligere. Her skal vi bruke et tre med følgende noder:

```
public class TreNode {
    public String navn;
    public TreNode forelder;
    public LinkedList<TreNode> barn = new LinkedList<>();

    public TreNode (String navn, TreNode forelder){
        this.navn = navn;
        this.forelder = forelder;
    }
    @Override
    public String toString(){
      //TODO Legg til kode her
    }
}
```

En trenode har altså et navn, en forelder og en liste med barn. Hvis listen med barn er tom, så er noden et blad, og hvis forelder er null, så er noden en rotnode.

Når vi skal skrive ute en node, vil vi ha med alle foreldrene også – helt opp til rotnoden. Vi vet jo ikke hvor langt det er, men rekursivt løser vi det slik:

1. forelder er null: navn
2. forelder er ikke null: forelder.toString() + navn

Hvis vi har en node som ikke er rotnode, vil forelder ha verdi. Da kaller vi forelder sin toString før vi legger til nodens eget navn. Forelder sin toString vil igjen kalle *sin* forelder sin toString før det legger til sitt eget navn osv. helt til vi finner rotnoden som ikke har forelder og bare returnerer sitt navn. Resultatet er alle navn for nodene fra roten ned til og med noden selv.

```
@Override
public String toString(){
    //Rekursiv toString opp til roten ("bread crumbs")
    if (forelder==null) {
        return navn;
    }
    return forelder.toString() + " - " + navn;
}
```

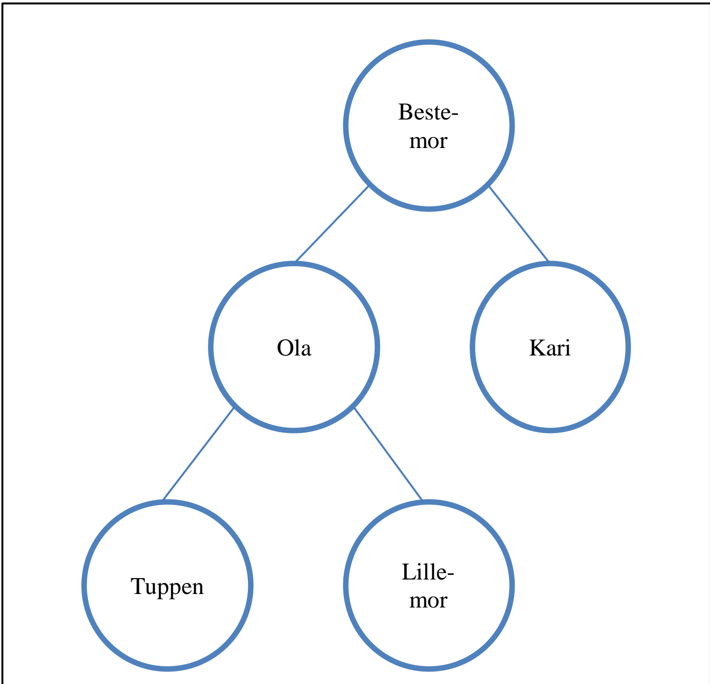
For å få skrevet ut alle nodene, bruker vi en annen rekursjon *traverser(node)*. Den er definert slik:

1. Hvis bladnode (ingen barn): nodens toString
2. Ellers for alle barn: traverser(barn)

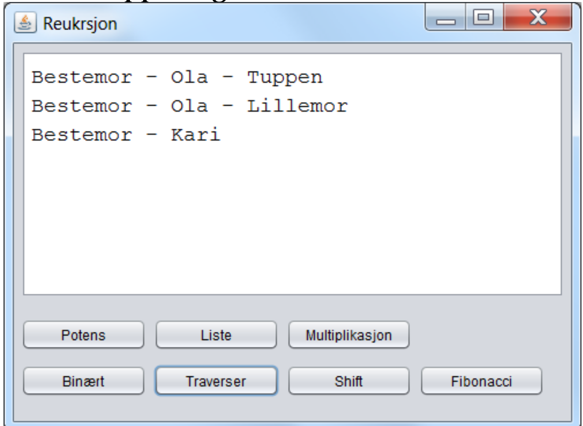
Regel 2 er riktignok iterativ men den kaller jo *traverser* rekursivt en gang for hvert barn.

```
private void traverser(TreNode node){
    if (node.barn.isEmpty()) { //bladnode
        vis.append(node.toString() + "\n");
    }
    else {
        for (TreNode n: node.barn){
          traverser (n);
        }
    }
}
```

Anta at vi har følgende tre:



Da vil rekursjonen *traverser* starte med roten (Bestemor) som har to barn, for hvert barn traverser den videre til dennes barn osv. og kommer etter hvert til en node (blad) uten barn. Da returneres denne nodens toString som klatrer opp igjen til sin forelder osv. helt opp til roten og navnene returneres fra toppen og ned. Resultatet ser slik ut:



Eksempel: Shift-operasjon

En shift-operasjon er en bit-vis operasjon der hver eneste bit i en byte flyttes en plass til venstre eller høyre. Biten som står i den ene enden "forsvinner" og det fylles på med biten 0 i den andre enden.

Anta at bitmønsteret ser slik ut:

1	0	0	1	1	0	0	1

Etter shift én til høyre blir resultatet:

0 →

0	1	0	0	1	1	0	0

→ 1

I Java skrives det med operatoren >> f.eks. *tall*>>*x* der *x* angir antall plasser bitmønsteret skal flyttes mot høyre. (Den motsatte operatoren finnes også.)

En AND operasjon mellom to tall virker slik at hver bit i det ene tallet sammenholdes med tilsvarende bit i det andre. Hvis begge er 1 blir resultatet 1, ellers 0, f.eks.

153_{10}	1	0	0	1	1	0	0	1
AND 15_{10}	0	0	0	0	1	1	1	1
= 9_{10}	0	0	0	0	1	0	0	1

Effekten er at det andre tallet virker som et "filter" som viser hvilke av de fire siste bitene som er "på" (dvs at de er 1).

Forlengs er det enkelt å skrive et tall binært – jeg bruker metoden *Integer.toBinaryString(tall)*. Jeg skal skrive det ut baklengs.

Ved å bruke 1 (=00000001) som filter kan vi se om siste bit er "på" (altså 1) eller "av" (altså 0). Dette skal jeg nå anvende. Jeg kaller funksjonen *shift(tall)*:

1. tall=0: Gjør ingen ting – ferdig.
2. tall>0: Noter siste siffer. Shift *tall* én til høyre. Kall *shift(tall)*.

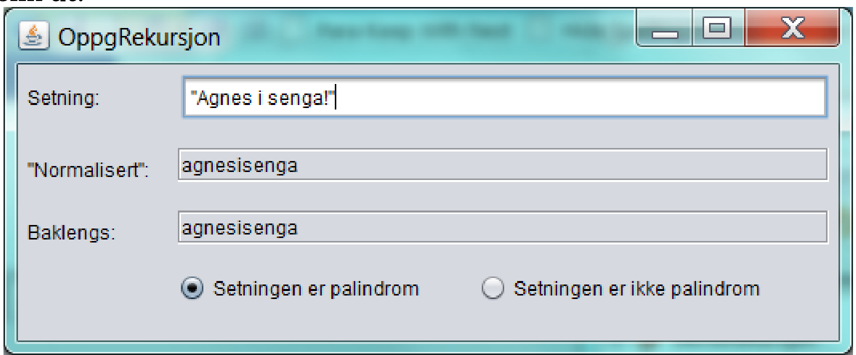
```
private void shift (int tall){
    if (tall > 0){
        int siffer = tall & 1; //Bruker AND med "filteret" 00000001
        vis.append(Integer.toString(siffer));
        tall = tall >> 1; //skifter en posisjon til høyre
        shift(tall);
    }
}
```

Oppgave til kapittel 7

Dette er en oppgave mest for moro – og du har løst liknende før (i Visual Basic). Du skal sjekke om en streng som brukeren skriver inn er et "palindrom". Palindrom defineres som en streng som det samme enten den leses forlengs eller baklengs. Vi ser bort fra alle mellomrom og tegnsetting (dvs alt annet enn bokstavene) og skiller ikke mellom store og små bokstaver.

Du skal "snu" setningen *rekursivt*.

Skjemaet ser slik ut:



Brukeren taster inn en setning i tekstfeltet. Når det rykkes ENTER, fylles resten ut.

Noen vink:

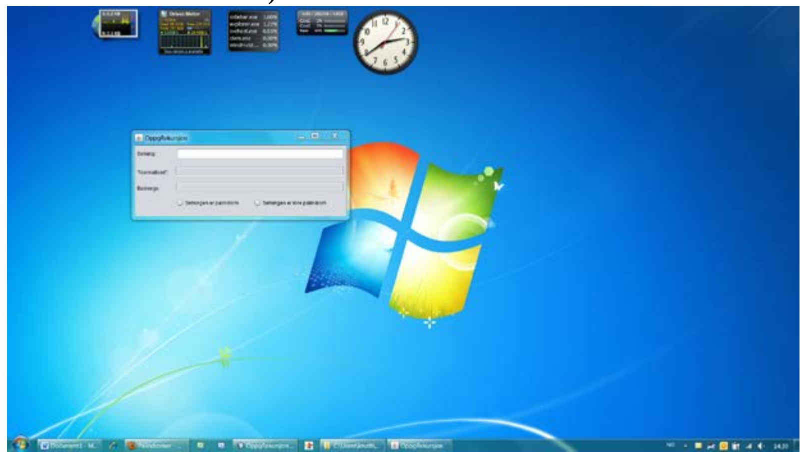
1. Her er det helt uaktuelt med kontrollklasse – alt skjer i skjemaet.
2. Du må knytte hendelsen *keyTyped* til tekstfeltet. Da sjekker du om ENTER ble trykket, f.eks. med
   if (evt.getKeyChar()=='\n') //brukeren trykket ENTER
3. "Normalisering" innebærer å fjerne alt som ikke er bokstaver. Du kan bruke streng-metoden *replaceAll("^[a-zæøå_0-9]","")* som erstatter alt som ikke er blant de nevnte tegnene med en tom streng. Husk også å gjøre om til bare små bokstaver *først*.

4. Den rekursive funksjonen som snur en streng, kan kalles *snu(streng)* og beskrives slik:
    a. Hvis strengen er tom eller ett tegn så returner strengen
    b. Ellers så returner siste tegn i strengen etterfulgt av resten av strengen (dvs. unntatt siste tegn) snudd.
       Eksempel: Strengen er "abc", da returneres "c" + snu("ab").
5. Avslutt med å sette strengen normalisert og omvendt inn i skjemaet og sett riktig radioknapp.

Du finner mange, morsomme palindromer på Internett, f.eks. følgende fra en litt "fuktig" fest: *Rolf Are vurderer om Arons ni drag i gardinsnora morer edru Vera Flor.* Eller denne, som for sikkerhets skyld også er rekursiv: *Ella redder Kim, Kim redder Mik, Mik redder alle.*

Tall kan også være palindromer som *2002* (og det er lenge til neste palindrome årstall – det er ikke mange som opplever mer enn ett). For øvrig brukes ordet *palindrom* både om DNA og gikt.

Ekstra: Prøv om du denne gangen kan få vinduet til å dukke opp litt inne på desktop (og ikke øverst til venstre som er default):



Kapittel 8 – Databaser

I dette kapittelet skal jeg vise hvordan man kan programmere Java mot en database. Jeg ser på ODBC/JDBC og databasene MySQL, Oracle og Java DB ("derby")

ODBC eller JDBC?

ODBC ("Open Data Base Connectivity11) er bibliotek utviklet av Microsoft i programspraket C. Det er laget for a knyttet programmer til en database omtrent som programmet knytter seg til en printer. De kaller derfor ODBC en *driver*. ODBC er komplekst, omfattende og dermed fleksibelt. ODBC finnes for de fleste databaser, programmeringsspråk og operativsystemer. Det er database-leverandøren som lager ODBC for sin database, men de vil alle fø1ge samme lest og ha de samme metodene. ODBC tar seg av omgjøring av de standardiserte kallene og datatypene i din applikasjon, til tilsvarende kall og datatyper som passer for databasen. Siden ODBC er skrevet i C med sine egne datatyper, kan det bli mye "oversettelse" frem og tilbake (f.eks. fra Java til C til Oracle og retur). Det må derfor lages en ODBC for hvert programmeringsspråk. Til gjengjeld kan ODBC da brukes med mange programmeringsspråk.

JDBC ("Java Data Base Connectivity API") er biblioteket utviklet i Java av Sun. Det er svært parallelt til ODBC, men er bevisst holdt enklere/mindre og derved ikke sa fleksibelt. Også JDBC finnes for de fleste databaser og operativsystemer. Det blir mindre problemer med datatyper, da JDBC skrives i Java. På den annen side er det uaktuelt å bruke JDBC i andre programmeringsspråk enn Java. Siden Java kjører i en virtuell maskin, klarer det seg med en JDBC for hver database (og ikke en for hvert språk).

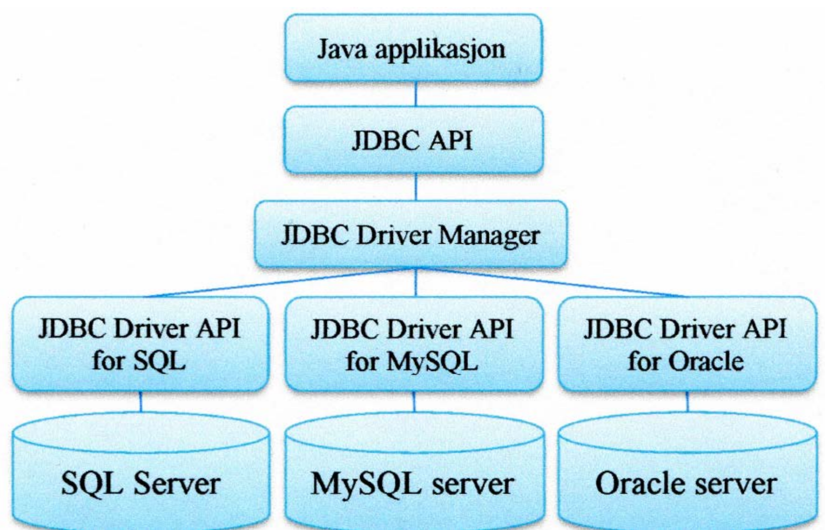
Endelig finnes det en *JDBC-ODBC* bridge samt en *ODBC-JDBC* bridge som gjør det mulig å bruke den ene i applikasjonen men den andre for tilknytningen til databasen. Det foregår altså en "oversettelse" fra den ene til den andre. På den måten kan man knytte seg til en database som bare tilbyr ODBC men allikevel anvende JDBC i applikasjonen. Dette er ikke så aktuelt, fordi de fleste databaseleverandørene tilbyr begge. De er også tregere fordi det blir to "oversettelser".

I valget mellom disse, må man se på den tenkte bruken. Fordi ODBC har flere muligheter, bør man kanskje velge det for Windows applikasjoner. ODBC kan også brukes sammen med mange språk, inkludert Java. JDBC passer bare hvis du skal programmere i Java og det er aktuelt å bruke systemet på mange plattformer.

Forbindelsen mellom din applikasjon og databasen skjer gjennom samlinger av klasser – i biblioteker. Nærmest applikasjonen ligger JDBC API som er helt generisk og ikke knyttet til noen bestemt database - den følger med Java. Koblingen til databasen skjer videre gjennom JDBC Driver Manager. Den holder orden på driverne - det kan være flere av dem - og gjør det mulig å knytte applikasjonen til flere databaser. Det er Driver Manager som kobles til databasen med en *connection.* Siden JDBC API og Driver Manager er generiske, må objekter "oversette" kommandoen og datatypene til/fra den enkelte databasetype. "Oversettelsen" gjøres av forskjellige, tilpassede JDBC Drivere. JDBC Driver leveres derfor av hver enkelt databaseleverandør.

For ODBC blir det helt likt, bortsett fra at biblioteket både må passe for databasen og for ditt programmeringsspråk.

Se figur for JDBC – ODBC blir helt analogt. Med JDBC-ODBC Bridge vil det bli ytterligere ett lag, nemlig broen mellom JDBC API og en ODBC Driver Manager.



Vi skal her bruke JDBC, fordi den er enkel og passer best til Java.

1. MySQL

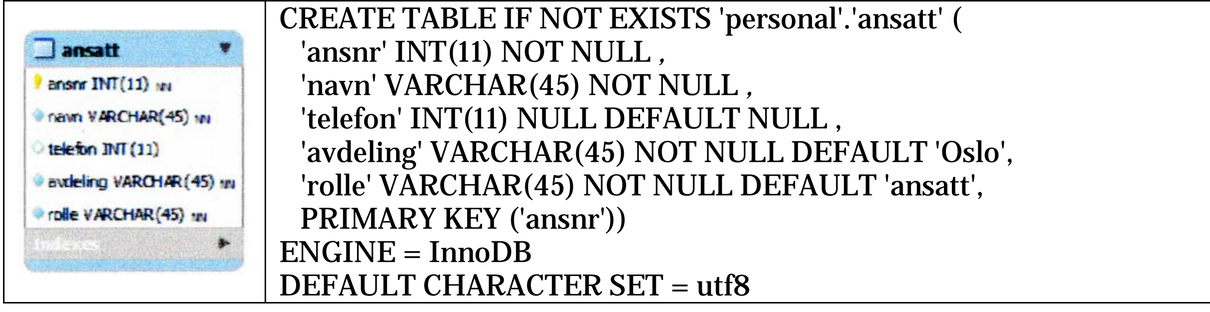
Du må for det første ha MySQL (Community) databasen installert, gjerne med en eksempeldatabase. Videre er det greit å bruke NetBeans og da har du også JDBC-biblioteket fra MySQL - det heter Connector J. Det finnes installasjonsfiler for Windows og andre OS som har med det meste. Du finner dem på http://dev.mysql.com/downloads/installer/

Dette er med:

- ✓ My SQL Server 5.5 GA
- ✓ MySQL Workbench 5.2 GA
- ✓ MySQL Connectors (.Net / ODBC / Java / C / C++) GA
- ✓ MySQL Notifier
- ✓ MySQL Samples and Examples 5.5
- ✓ MySQL Documentation 5.5

Under installasjonen kan man velge hva som skal installeres og om man vil ha 32 eller 64 bits versjonen. Denne har dere brukt flere ganger før og har antakelig fortsatt installert på deres egen maskin. Sannsynligvis har du allikevel bruk for å installere "Connectors".

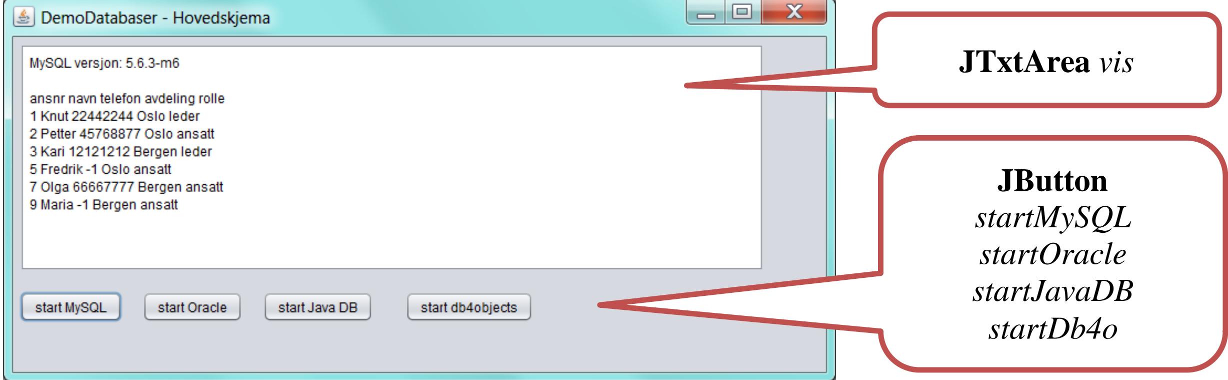
Mitt databaseskjema heter *Personal* og har tabellen *Ansatt* som jeg også har lagt inn noen data i:




```
CREATE TABLE IF NOT EXISTS 'personal'.'ansatt' (
  'ansnr' INT(11) NOT NULL ,
  'navn' VARCHAR(45) NOT NULL ,
  'telefon' INT(11) NULL DEFAULT NULL ,
  'avdeling' VARCHAR(45) NOT NULL DEFAULT 'Oslo',
  'rolle' VARCHAR(45) NOT NULL DEFAULT 'ansatt',
  PRIMARY KEY ('ansnr'))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8
```

Når du lager programmet, høyreklikker du på *Libraries* og legger til biblioteket *MySQL JDBC* Driver.

Mitt skjema (en *JFrame)* ser slik ut:



Jeg importerer *java.sql.** og *javax.swing**

Koden

Jeg starter med å åpne en *connection-objekt* med *Driver Manager*[30]. Den åpner databasen og må vite (1) protokoll, (2) databasetype, (3) adresse/port samt (4) skjemanavnet Til sammen kalles det *url*. Den vil variere med databasetype og ser annerledes ut for Oracle, SQL Server osv. Videre oppgir jeg brukernavn og passord. For MySQL ser *url* for databaseskjemaet "personal" slik ut:

```
jdbc:mysql://localhost:3306/personal
```

Connection-objektene kan også utføre andre operasjoner mot databasen, bade DML og DDL og den kan håndtere transaksjoner. Det varierer fra databasetype til databasetype hva som er lovlig.

Connection-objektet brukes deretter til å lage et Statement-objekt som kan utføre SQL-setninger mot databasen. Hvis de gir et resultat – det er en spørring – så returneres resultatet som et *ResultSet-objekt. ResultSet-objektet* vil da inneholde en mengde med rader, nummerert fra 1 (OBS!). Man blar til neste med metoden *next()* som både flytter til neste rad og returnerer *true* hvis det fantes en rad til.

I første omgang henter jeg bare versjonsnummeret fra databasen. Det vil være i første rad (det returneres bare en) og første kolonne - som også nummereres fra 1 (OBS!).

I neste omgang henter jeg alle radene i tabellen *ansatt.* For hver av dem finner jeg kolonneverdiene ved å oppgi kolonnenavnet. Man kan her også bruke kolonnenummer. *ResultSet-objektet* har også et *MetaData-objekt* som inneholder navnene på kolonnene, datatypene og annet. Dette bruker jeg til å skrive ut kolonnenavnene.

Jeg avslutter med å lukke forbindelsen. Databasen lukkes, databuffere fristilles osv. Vi kan ellers risikere å få minnelekkasje (minnet forblir opptatt etter at applikasjonen er lukket). Det understrekes sterkt i dokumentasjonen et dette er en reell risiko. Vær også oppmerksom på

---

[30] Mange steder på Internet står det at man må starte med å laste driveren. Koden for det ser slik ut: *Class forName(<driver>).* Fra og med versjon JDK 6 er dette unødvendig da driveren automatisk lastes når en forbindelse (connection) åpnes.

at *close()* ikke er *idempotent.* Det betyr at den ikke virker likt vær gang den brukes. Hvis *close()* brukes på en *connection* som allerede er lukket, er følgelig resultatet udefinert. (Hvis du ikke synes det er skummelt, så har du bare ikke erfart det ennå!) Derfor sjekker jeg at den ikke er lukket før jeg gjør det.

Nedenfor ser du den ferdige koden, og resultatet ved kjøring er gjengitt i bildet av skjemaet ovenfor. Merk spesielt at sql-setningene *ikke* skal avsluttes av semikolon. En fin ressurs for å lære seg mer er http://zetcode.com/databases/mysqljavatutorial/

```
private void startMySQLActionPerformed(java.awt.event.ActionEvent evt) {
    String url = "jdbc:mysql://localhost:3306/personal";
    String bruker = "Knut";
    String passord = JOptionPane.showInputDialog
        (rootPane, "Oppgi passord","Tunk"); //TODO: Fjerne forslaget
    Connection conn = null;
    Statement setning;
    ResultSet resultat;

    /* 1. Skap en forbindelse til databasen */
    try {
        conn =  DriverManager.getConnection
                (url, bruker, passord);
    }
    catch(Exception ex){
        JOptionPane.showMessageDialog
                (rootPane, ex.getClass() + "\n" + ex.getMessage());
    }

    /* 2. Kjør en spørring etter versjonsnummer */
    try{
         setning = conn.createStatement();
         resultat = setning.executeQuery("select version()");
         if (resultat.next()) {
            vis.setText("MySQL versjon: "
                    + resultat.getString(1) + "\n\n");
         }
    }
    catch (Exception ex) {
        JOptionPane.showMessageDialog
                (rootPane, ex.getClass() + "\n" + ex.getMessage());
    }

    /* 3. Kjør en spørring til ansatt-tabellen */
    try {
        setning = conn.createStatement();
        resultat = setning.executeQuery
                ("select * from ansatt order by ansnr");
        // Vis kolonnenavnene OBS Første kolonne har nummer 1
        for (int i=1;i<=resultat.getMetaData().getColumnCount();i++){
            vis.append(resultat.getMetaData().getColumnLabel(i) + " ");
        }
```

```
            //Vis alle dataene
            vis.append("\n");
            while (resultat.next()) {
                vis.append(Integer.toString(resultat.getInt("ansnr"))+" ");
                vis.append(resultat.getString("navn") + " ");
                vis.append(Integer.toString(resultat.getInt("telefon"))+"
");
                vis.append(resultat.getString("avdeling") + " ");
                vis.append(resultat.getString("rolle") + " ");
                vis.append("\n");
            }
        } catch (Exception ex) {
            JOptionPane.showMessageDialog
                    (rootPane, ex.getClass() + "\n" + ex.getMessage());
        }

        /* 4. Lukk forbindelsen */
        try {
            if (conn != null) {
                conn.close();
            }
        }
        catch (Exception ex) {
            JOptionPane.showMessageDialog
                    (rootPane, ex.getClass() + "\n" + ex.getMessage());
        }
    }
```

Som du ser er hver enkelt operasjon ganske grei.

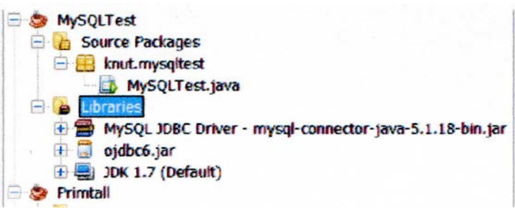
2. Oracle database

OBS! Vår Oracle kjører på en virtuell server som kan være treg. Forvent ventetid!

Du må ha biblioteket for Oracle database versjon *Oracle Database llg Enterprise Edition Release 11.10.6.0 - 64bit Production*. Den finner du på http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html. Velg *ojdbc6.jar* for Oracle 11g. (Senere Oracleversjoner støtter også nyere versjoner av *OJDBC*.)

Plasser gjerne *jar*-filen i subkatalogen *C:\Program Files\Java\jre8\lib\ext* eller tilsvarende på din maskin. Det er standard for ekstra biblioteker.

Når du lager programmet, høyreklikker du på *Libraries* og legger til *Jar/Folder* og refererer til *jar*-filen *ojdbc6.jar*.



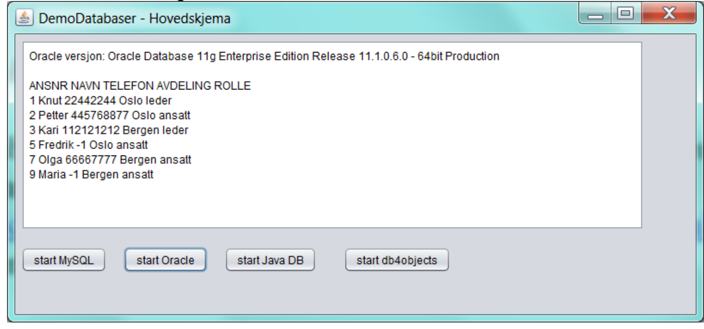
Connection-strengen har en litt annen syntaks her enn for MySQL. For Oracle heter den for tiden[31]

```
jdbc:oracle:thin:@158.36.15.19:1521:HIBURO
```

med ip-adressen til den databasen vi bruker (@158.36.15.19), porten (1521) og system id (HIBURO). Her oppgir vi ikke noe skjemanavn, da Oracle oppretter et skjema for hver bruker isteden. System ID - også kalt SID – identifiserer hver enkelt Oracle database hvis det kjører flere på en maskin.

Ellers blir programmet helt likt, unntatt spørringen som finner versjonsnummeret for databasen.

Fordi Oracle har store bokstaver på kolonnenavn, blir utskriften litt annerledes:



Programmet for Oracle er gjengitt på neste side. Merk spesielt at sql-setningene *ikke* skal avsluttes av semikolon, slik Oracle selv vanligvis forlanger.

```
private void startOracleActionPerformed(java.awt.event.ActionEvent evt)
{
    String url = "jdbc:oracle:thin:@158.36.15.19:1521:HIBURO";
    String bruker = "KnutH";
    String passord = JOptionPane.showInputDialog
        (rootPane, "Oppgi passord","Tunk"); //TODO: Fjerne forslaget
    Connection conn = null;
    Statement setning;
    ResultSet resultat;
```

Punkt 1 (skape en forbindelse) er som for MySQL. Deretter følger:

```
    /* 2. Kjør en spørring etter versjonsnummer */
    try{
         setning = conn.createStatement();
         resultat = setning.executeQuery
                ("select * from v$version where banner like 'Oracle%'");
         if (resultat.next()) {
            vis.setText("MySQL versjon: "
                    + resultat.getString(1) + "\n\n");
         }
    }
    catch (Exception ex) {
        JOptionPane.showMessageDialog
                (rootPane, ex.getClass() + "\n" + ex.getMessage());
    }
```

Punkt 3 (spørring til ansatt-tabellen) og punkt 4 (lukking av forbindelsen) er nøyaktig som for MySQL og gjentas ikke her. Det skyldes at JDBC API er den samme for begge - det er bare driveren som byttes ut.

[31] Oracle-serveren er under flytting når dette skrives. Da vil antakelig både databasens ip-adresse og navn endres. Spør din faglærer om hva som er aktuelt når du skal prøve dette.

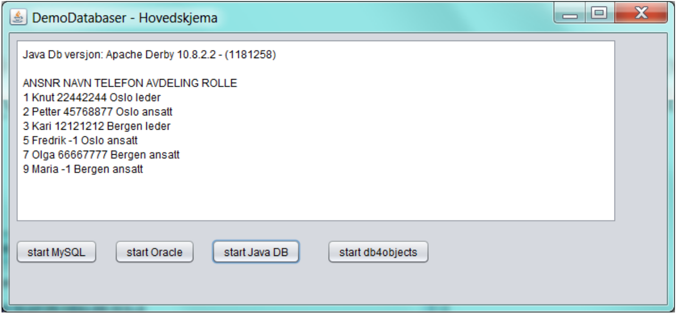
3. Java DB database ("derby")

Java JDK har også sin egen, innebygde database. Den brukes på samme måte som MySQL og Oracle, men i motsetning til dem, definerer man databasen, tabeller, brukere, views osv. *fra programmet,* dvs. "embedded". Det er (nesten) ingen separat DBMS som man kan få tilgang til utenom Java-applikasjonen - alt programmeres. I mange sammenhenger er dette en betydelig ulempe, men det er enkelt. På mange måter virker det som filer, men her kan vi ha flere tabeller med sammenheng mellom postene og får integritetskontroll (referanse- og domeneintegritet). *Derby* kan dessuten også kjøres på en server mot flere klienter samtidig. Her bruker vi imidlertid *derby* lokalt med bare én klient.

Biblioteket heter *derby.jar* og det finner du under JDK-katalogen i submappen *db.* Du må knytte programmet til dette biblioteket med høyreklikk på *Libraries.*

Connection-strengen ser slik ut for Java DB: *"jdbc:derby:ansatt;create=true";* Som vanlig er det henvist til API, til databasetypen og til databasenavnet. Strengen kan ha parametre adskilt med semikolon og her har jeg lagt til *create=true.* Det innebærer at databasen skal skapes hvis den ikke allerede finnes. Det kan være andre parametre også.

Databasen skapes først når forbindelsen skapes. Det lages da en submappe under programmappen og den blir hetende det samme som databasenavnet - her *ansatt.* Der lagres dataene og alle hendelser logges. Dataene lagres i filer på maskinlesbar form (ikke forståelig for mennesker).



Siden vi ikke kan lage databasen og legge til tabeller og data annet enn gjennom et annet program, har jeg i dette programmet to ekstra punkter: Generere en tabell og legge inn noen data. Å skape tabellen gjøres med *create table* som vanlig. Jeg passer da på å bruke variabeltyper som Java DB kjenner, ellers er det nokså rett frem (se koden nedenfor). For at programmet skal gå an å kjøre flere ganger, har jeg her valgt å droppe tabellen og skape den pånytt for hver start. Jeg ville nok ellers ha laget et eget program som oppretter databasen og nødvendige tabeller og kjørt dette programmet bare en gang.

Dataene legges inn med vanlige insert-setninger. Sql-setningene skal ikke avsluttes med semikolon.

Programmet ser slik ut:

```
private void startJavaDBActionPerformed(java.awt.event.ActionEvent evt)
{
    String url = "jdbc:derby:ansatt;create=true"; //skaper basen
     String bruker = "KnutH";
     String passord = JOptionPane.showInputDialog
         (rootPane, "Oppgi passord","Tunk"); //TODO: Fjerne forslaget
     Connection conn = null;
     Statement setning;
     ResultSet resultat;
```

Punkt 1 (skape forbindelse til databasen) er som tidligere. Deretter kommer to nye punkter:

```
        /*1A EKSTRA. Slett og gjenskap en tabell */
        try {
            String dropString = "drop table ansatt";
            String createString = "create table ansatt "
                 + "("
                 + "ansnr int primary key,"
                 + "navn varchar(45),"
                 + "telefon int,"
                 + "avdeling varchar(45) not null default 'Oslo',"
                 + "rolle varchar(45) not null default 'ansatt'"
                 + ")";
            setning = conn.createStatement();
            setning.execute(dropString);
            setning.execute(createString);
        } catch (Exception ex) {
            JOptionPane.showMessageDialog
                    (rootPane, ex.getClass() + "\n" + ex.getMessage());
    }
        /*1B EKSTRA. Legg inn noen ansatte */
        try {
            String insertString = "insert into ansatt values"
                    + "(1,'Knut',22442244,'Oslo','leder'),"
                    + "(2,'Petter',45768877,'Oslo','ansatt'),"
                    + "(3,'Kari',12121212,'Bergen','leder'),"
                    + "(5,'Fredrik',-1,'Oslo','ansatt'),"
                    + "(7,'Olga',66667777,'Bergen','ansatt'),"
                    + "(9,'Maria',-1,'Bergen','ansatt')";
            setning = conn.createStatement();
            setning.execute(insertString);
        }
        catch (Exception ex) {
            JOptionPane.showMessageDialog
                    (rootPane, ex.getClass() + "\n" + ex.getMessage());
        }
```

Versjonen kan antakelig finnes med en spørring, men jeg finner ingenting om det, verken i manualen eller på Internett. Jeg gjør det derfor annerledes her enn med de to andre databasene. Jeg benytter meg da isteden av connection-objektets metadata som inneholder opplysninger om databasen:

```
        /* 2. Kjør en spørring etter versjonsnummer */
        try{
            String versjon = conn.getMetaData().getDatabaseProductName()
                     + " " + conn.getMetaData().getDatabaseProductVersion();

            vis.setText("Java Db versjon: "
                        + versjon + "\n\n");
        }
        catch (Exception ex) {
            JOptionPane.showMessageDialog
                    (rootPane, ex.getClass() + "\n" + ex.getMessage());
        }
```

Punktene 3 (spørring mot ansatt-tabellen) og punkt 4 (lukking av forbindelsen) er som tidligere.

Sql-setninger for Statement (uansett database)

Disse finner du i dokumentasjonen *JavaDoc* som vanlig. Du vil finne at man kan utføre flere setninger som skript, at man kan benytte transaksjoner og utføre setninger som ikke returnerer rader – typisk DML og DDL-setninger.

Hvis eksekveringshastigheten virkelig er kritisk, kan man prekompilere sql-setningene, men det tar vi ikke med her (*PreparedStatement*).

Andre databaser

Det finnes mange andre databaser med JDBC API, men ovenfor har jeg vist at de vil bli relativt like og fø1gelig avslutter jeg omtalen av dem her.

Ekstra: Kort oversikt over gangen i koden

1. Skap et Connection-objekt (driveren lastes automatisk):
   ```
   Connection conn =  DriverManager.getConnection
     (url, bruker, passord);
   ```
   url er en streng som angir hvilket bibliotek som skal brukes (JDBC), hvilken driver (MySQL, Oracle, derby osv.) og hvor database er å finne (syntaksen varierer fra driver til driver), f.eks.
   ```
   String url = "jdbc:mysql://localhost:3306/personal";
   ```

2. Skap et *Statement*-objekt:
   ```
   Statement setning = conn.createStatement();
   ```

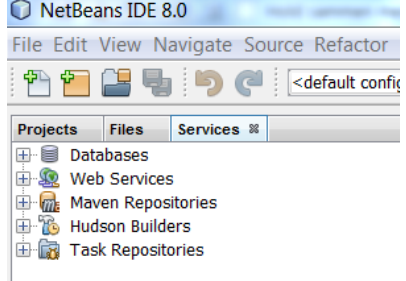
3. Bruk *Statement*-objektet til noe, f.eks. *execute*, *ececuteQuery* eller *executeUpdate*:
   ```
   boolean ok = setning.execute('insert into personal values
       (555, 'Knut Hansson', 68);');

   int antall = setning.executeUpdate
       ('delete from personal where persnr = 555;');

   ResultSet resultat = setning.executeQuery
       ('select * from personal where lonnstrinn > 70;');
   ```
   SQL-syntaksen følger naturligvis syntaksen til den aktuelle databasen.

4. Bruk evt. *ResultSet*-objektet til noe, f.eks. å hente kolonneoverskrifter, og i alle fall rader. *ResultSet*-objektet er ikke en samling, men har en cursor som peker til én av radene som returneres cursoren er først *null*). *resultat.next()* er true hvis det finnes flere rader og samtidig flytter det da cursoren én rad videre, så den egner seg for while-løkker.

5. Gjenta 3 og 4 og lukk så forbindelsen til slutt:
   ```
   if (conn != null) conn.close();
   ```
   OBS! Du må bare lukke forbindelser som faktisk er åpne!

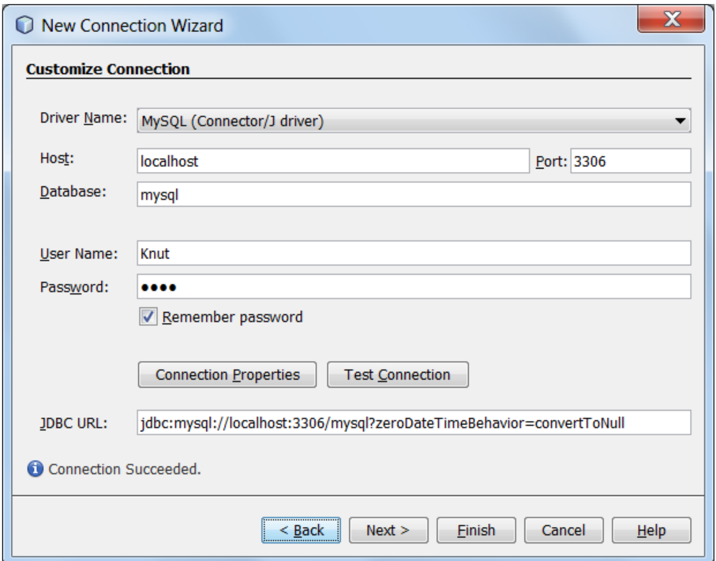
Ekstra: Bruke NetBeans Database Explorer

Innebygget i NetBeans er det verktøy som likner MySQL Explorer i (svært) enkel utgave. I vinduet til venstre står vanligvis prosjektene listet opp. Der finnes det imidlertid to faner til, nemlig "Files" og "Services". Her er "Services" åpnet:

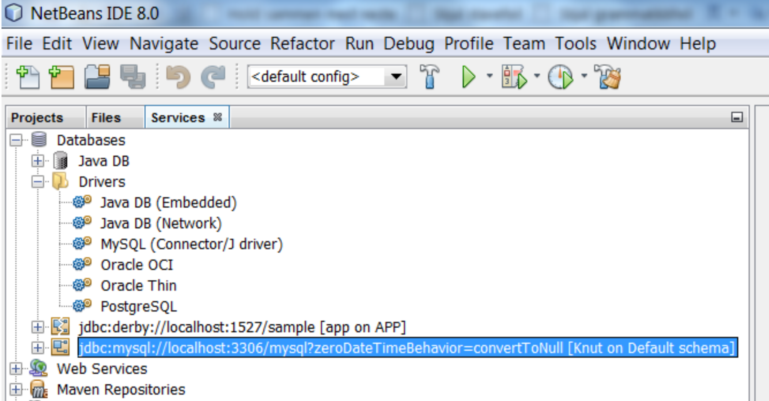


Det første på listen som vises, er "Databases". Den kan åpnes og gir da mulighet for å tilknytte seg databaser gjennom en rekke, forskjellige drivere.

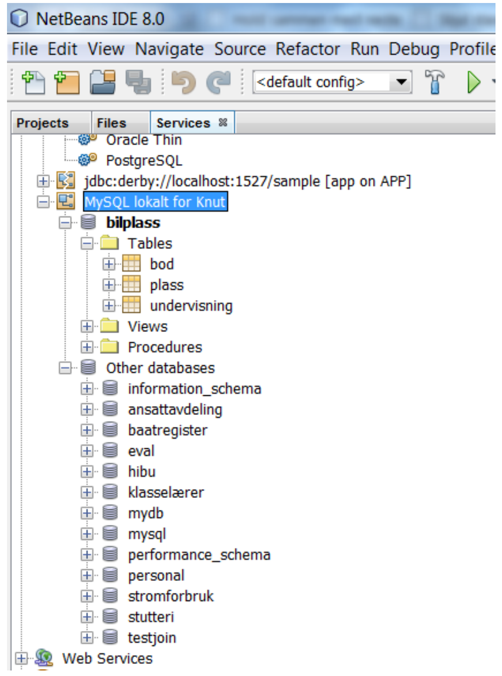
Jeg høyreklikker f.eks. "MySQL" og velger "Connect using...". I dialogboksen er det meste ferdig utfylt, men jeg oppgir brukernavnet mitt (i MySQL) og passordet før jeg tester forbindelsen.



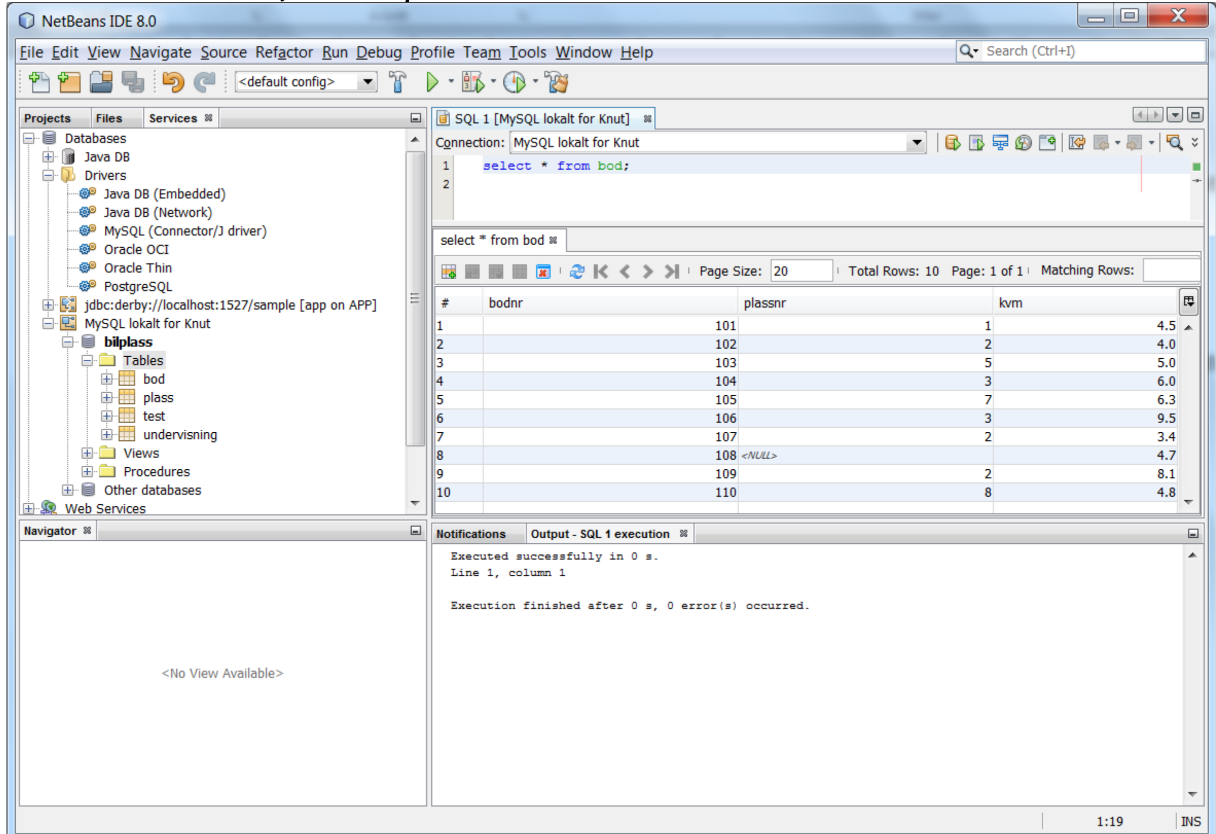
Her fikk jeg forbindelsen greit. Jeg klikker så "Finish" og får en ny forbindelse:



Forbindelsen har et svært langt navn, men det kunne jeg endret ved å klikke "Next" istedenfor "Finish" i forrige trinn eller etterpå i forbindelsens egenskaper (høyreklikk). Dobbeltklikker jeg på forbindelsen – etter evt. å ha endret navnet på den - får jeg se alle databaseskjemaene og tabellene mine. "Bilplass" er satt som default skjema (et valg ved høyreklikk).



Nå kan jeg se data, endre data og skjema, kjøre spørringer – kort sagt manipulere databasen som i MySQL Workbench. Høyreklikk på en "Tables" og velg "Execute Command" så åpnes et SQL kommandovindu. Du ser hvordan det likner litt på *MySQL Workbench* og mye på den tilsvarende *Oracle SQL Developer*.



#	bodnr	plassnr	kvm
1	101	1	4.5
2	102	2	4.0
3	103	5	5.0
4	104	3	6.0
5	105	7	6.3
6	106	3	9.5
7	107	2	3.4
8	108	<NULL>	4.7
9	109	2	8.1
10	110	8	4.8

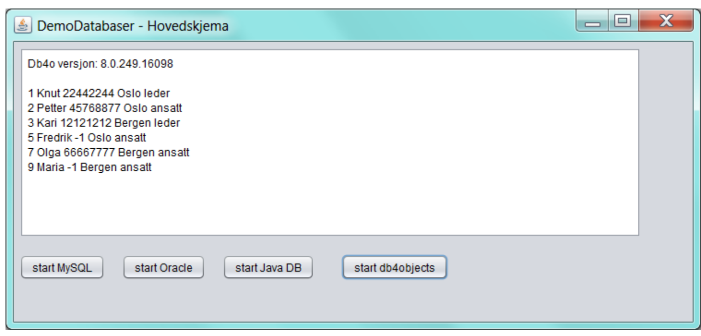
Dette kan jo være veldig kjekt når man jobber mot en database, siden man slipper å åpne flere programmer.

Ekstra: Objektorientert database

Et kjent problem er "the impedance mismatch" mellom et OOP og en relasjonsdatabase. Som dere har sett tidligere, er mappingen mellom de er kompleks og kan fort bli gal.

db4objects (db4o)

I eksemplet har jeg vist bruk av den *objektorienterte* databasen *db4o* fra *Versant*. Denne databasen er en undergruppe av graf-database. Bruken av db4o er svært enkel å programmere, fordi konvertering mellom datalageret og objektene samt assosiasjonene mellom dem skjer automatisk. Den er også svært rask.




```
//Logg inn
String bruker = "KnutH";
String passord = JOptionPane.showInputDialog
        (rootPane, "Oppgi passord","Tunk"); //TODO: Fjerne forslaget
final ObjectContainer odb
        = Db4oClientServer.openClient
        ("localhost", 8080, bruker, passord);
```

Legge inn noen objekter:

```
//Lag noen objekter og lagre dem
try {
    Ansatt nyAnsatt = new Ansatt (1,"Knut",22442244,"Oslo","leder");
    odb.store(nyAnsatt);
    nyAnsatt = new Ansatt(2,"Petter",45768877,"Oslo","ansatt");
    odb.store(nyAnsatt);
    nyAnsatt = new Ansatt(3, "Kari", 12121212, "Bergen", "leder");
    odb.store(nyAnsatt);
    nyAnsatt = new Ansatt(5, "Fredrik", -1, "Oslo", "ansatt");
    odb.store(nyAnsatt);
    nyAnsatt = new Ansatt(7, "Olga", 66667777, "Bergen", "ansatt");
    odb.store(nyAnsatt);
    nyAnsatt = new Ansatt(9, "Maria", -1, "Bergen", "ansatt");
    odb.store(nyAnsatt);
    odb.commit();
} catch (Exception ex) {
    JOptionPane.showMessageDialog
            (rootPane, ex.getClass() + "\n" + ex.getMessage());
}
```

Legg merke til hvor enkel lagringen er – du bruker bare metoden *store*. Det er ingen konvertering som du programmerer selv. Tilsvarende slettes et objekt i databasen med *delete*. Oppdatering av objekt i databasen er ikke aktuelt – du henter objektet til RAM, endrer det der og lagrer det igjen.

Du kan hente objekter med *query by example*. Det innebærer at du lager en "mal" for hva du vil ha (her en generell ansatt). QBE vil returnere alle objekter i databasen som passer til malen:

```
ObjectSet<Ansatt> alle = odb.queryByExample(Ansatt.class);
```

Mer fleksibelt er å bruke *query*. Den skal ha et Predicate-objekt der metoden *match* må overstyres. Det er den som bestemmer hva som returneres (alle objekter i databasen som har *match* lik *true* blir returnert):

```
//Hent alle ansatte med native query
Predicate<Ansatt> predikat = new Predicate<Ansatt>(){
    @Override
    public boolean match (Ansatt ansatt){
        return true; //returnerer alle denne gang
    }
};
ObjectSet <Ansatt> alle = odb.query(predikat);
```

Dette kalles *native query* fordi koden er skrevet i vertsspråket (her Java). Dermed blir koden kontrollert ved kompilering og typekontroll gjelder.

Hvis du skulle få lyst til å prøve deg selv, er *db4o* svært enkel å installere. Se nettsidene.

Oppgave til kapittel 8

Inspirert av databasens navn, skal vi registrere travhester og løp de har deltatt i. Programmet skal brukes av en som spiller på travhester. Dataene skal lagres i en Derby database. Oppgaven vil lære deg håndtering av databasen herunder spesielt Derby-basens syntaks og datatyper, og sikkert friske opp litt SQL. Vi skal ikke bruke objekter for hestene og løpene denne gang.

For hver hest registrerer vi hestens regnr (ID), hestens navn, eiers navn og om hesten er varm- eller kaldblods (*boolean varmblods*). For hvert løp som hesten deltar i, registrerer vi dato, distanse ("Kort", "Middels" eller "Lang") og plassering i løpet. Du kan trygt regne med at hesten bare deltar i ett løp pr dag (slik at hestenummer + dato er ID).

Fyll komboboksen for valg av løpslengde fra en *Enum*. Prøv å gjør det slik at man senere kan legge til flere løpslengder i *Enum* uten å endre koden i hovedprogrammet. **Vink**: Iterer over alle enumens *values()* og legg dem til komboboksen med *addItem*.

Forenklinger:

- ✓ Du kan nøye deg med å *registrere* hester og løp, og behøver ikke endre/slette.
- ✓ Lagre datoen som en ren streng uten noen kontroll.
- ✓ Hvis du ikke får til *Enum* så fyll heller komboboksen hardkodet i designmodus (komboboksens *modell*). Men OBS! *Enum* er pensum!
- ✓ Du kan hardkode brukernavn og passord.

Det skal være mulig å ta ut følgende rapporter, som vises i en meldingsboks:

1. Hvilke hester som er registrert, sortert etter navn.
2. Alle hester sortert etter antall løp de har deltatt sortert med de som har flest løp øverst.
3. Bare hester som har deltatt i kort løp sortert etter beste plassering i slike løp.

Husk at du må lage et eget program eller egen del av programmet som bare kjøres én gang, slik at du får opprettet Derby-databasen med de nødvendige tabeller o.l. Da kan du samtidig enkelt legge inn noen hester. (Du kan lage denne delen som egen programklasse i samme prosjekt og sette *Run*-egenskapen i prosjektets *Properties* til dette skjemaet før du kjører én gang.)

Forslag til databasemodell:

```
hest (regnr int,navn varchar(30),eier varchar(30),varmblods boolean)
løp  (*regnr int,dato varchar(20),distanse varchar(10),plassering int)
```

Ingen verdier får være *null*, *varmblods* har defaultverdi *false* og *plassering* skal være minst 1.

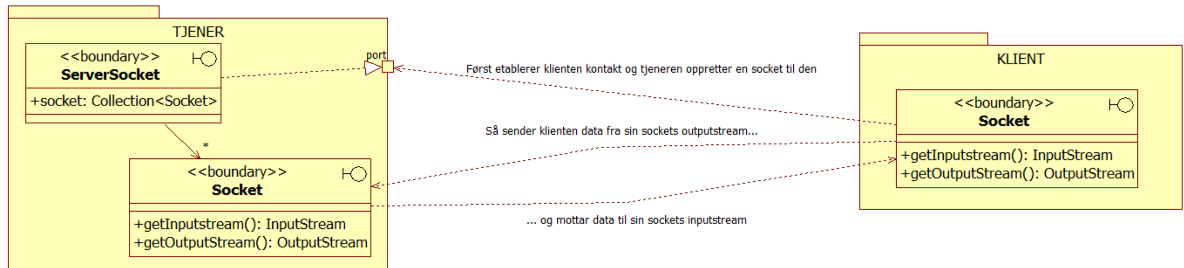
Lag grensesnittet selv, helst med en *JTabbedPane* med én fane for registrering av hest, én for registrering av løpsresultat og én for rapporter. (Du trenger mer trening i akkurat dette.) Lag også en singleton kontrollklasse denne gang – selv om det ikke har så mye for seg uten flere grensesnitt.

Kapittel 9 – Klient-tjener

Sockets

Java har ferdige klasser for klient-tjener kommunikasjon kalt *sockets*. Det er to typer: *Socket* både for klient og tjener og *ServerSocket* bare for tjenere.

Idéen er at ditt program på klientsiden, skaper en *Socket* som kontakter en *ServerSocket* over TCP/IP (eller UDP – en annen, enklere og mer usikker kommunikasjonsprotokoll). *ServerSocket* lytter på en bestemt *port*. Hvis kontakten godkjennes, skaper tjeneren en ny *Socket* som tar seg av kontakten med klienten i egen tråd. Den opprinnelige *ServerSocket* fortsetter å lytte etter flere klienter som ønsker kontakt.



Såsnart kontakt er opprettet, skjer kommunikasjonen mellom to like *sockets*-objekter. Begge sender data gjennom sin *outputstream* og mottar data gjennom sin *inputstream*.

Dette er vist dynamisk i presentasjonen Sockets animert.ppsx.

Note: Det kan være et problem at mens serverens socket kommuniserer med en klient så "henger" serveren og kan ikke ta imot henvendelser fra andre klienter. I et flerbrukermiljø er det uakseptabelt. Problemet oppstår fordi serversocket skaper en ny socket og sender en synkron melding til den. Da må serversocket vente på svar fra socket, dvs. til klienten er ferdig med kommunikasjonen.

Løsningen på dette er at serversocket starter socket *i en egen tråd*. Da kan den jobbe parallelt med at serversocket mottar henvendelser fra andre klienter. (Tråder ser vi på i et senere kapittel så her ser vi bort fra dette problemet.) Serveren sies da å være *concurrent* ("samtidig"). Du kan lese mer om dette under punkt 7.10 på http://homepages.feis.herts.ac.uk/~comqrgd/docs/network-notes/network-notes-node9.html

Note 2: Kommunikasjon med en nettside over web med en URL gjøres enklere på andre, mer høynivå måter. Da brukes også *sockets* til implementeringen, men det er skjult for programmereren. *Når vi bruker sockets er det altså for å kommunisere mellom programmer*. For at vi skal kunne bruke *Java sockets*, må begge programmene være skrevet i Java.

Ekstra: CORBA
Hvis programmene er skrevet i forskjellige programmeringsspråk, brukes isteden *CORBA*. Da kommuniserer objekter på den ene siden med objekter på den andre, f.eks. kan studentobjekter på klienten sende synkroniserte meldinger til kursobjekter på tjeneren, uavhengig av programmeringsspråk og maskinmiljø (så lenge CORBA finnes for språket). I CORBA heter *socket*-objektene ORB (*Object Request Broker*) og det må skrives en definisjon av grensesnittet mellom dem i et eget språk IDL (Interface Definition Language). De to programmene kompileres sammen med grensesnittbeskrivelsen og det skapes *stubs* på klientsiden og *skeletons* på den andre. Deretter kan objektene kommunisere direkte med hverandre innenfor den protokollen som er beskrevet i grensesnittet som om de

utgjorde ett program på én maskin. Dette er ikke pensum her, så interesserte henvises til Wikipedia.

Byte-strømmer mellom sockets

Note: Bytestrømmer har dere sett på tidligere, men da til/fra fil. Dette er derfor mye repetisjon. Det nye er at bytestrømmene oversendes via TCP mellom klient og tjener via sockets.

Sockets sender strømmer av bytes til hverandre. Det er jo *bytes* som pakkes og sendes over TCP. Det er altså ikke *objekter* som sendes. Hvis en *socket* skal overføre et objekt til en annen *socket*, må objektet *serialiseres* før det sendes og *deserialiseres* etter at det er mottatt (med klasser som passer til formålet) slik dere har gjort tidligere da dere lagret objekter på fil. Ofte kan man nøye seg med å sende/returnere tekster, f.eks. SQL-kommandoer med svar.

Det kan lages *sockets* som bedrer sikkerheten, med proxy og/eller med kryptering.

Skrive til en byte-strøm

Når du skal **skrive** kan du velge mellom flere klasser fra pakken *java.io*:

1. *BufferedWriter* skriver tegn. Skrivingen bufres for bedre effektivitet.
2. *OutputStreamWriter* konverterer tegn i et angitt tegnsett til bytes før skriving.
3. *DataOutputStream* skriver primitive datatyper (inkludert *String*) som bytes, f.eks. metodene *writeInt*, *writeBoolean*, *writeDouble* osv. Du må altså selv vite hvilken datatype som skal skrives (og det er vanligvis ikke noe problem).
4. *ObjectOutputStream* skriver alle typer primitive data, strenger og objekter. Objekter skrives med *writeObject(Object obj)*. Objektet som skrives må være *serializable*[32].

Som det forhåpentlig fremgår, bør man velge skriver som passer til leseren, ellers blir det mye konvertering og/eller feil. Bl.a. sendes det en *fileheader* som angir typen på strømmen. Hvis den ikke passer til leseren, gis det feil.

Lese fra en byte-strøm

Til å **lese** bytestrømmene, kan man også bruke forskjellige klasser fra *java.io*. Vanligvis oppgir du jo et filnavn som det skal leses fra og åpner filen. Her oppgir du en *InputStream* som filnavn og leser fra den. Hvilken klasse du bør anvende avhenger av hva som skal leses. Her er noen muligheter:

1. *BufferedReader* leser tegn. Lesingen bufres og det gir bedre effektivitet. Her finner vi f.eks. *readLine*.
2. *InputStreamReader* konverterer leste bytes til tegn[33] med et angitt tegnsett. Den kan kun lese *char* eller *char[]*.
3. *DataInputStream* leser bytes og tolker dem som primitive datatyper (inkludert *String*), f.eks. metodene *readInt*, *readBoolean*, *readDouble* osv. Du må altså selv vite hvilken datatype som kommer i strømmen. Du kan ikke lese objekter på denne måten.
4. *ObjectInputStream* leser alle typer primitive data, strenger og objekter. Objekter leses med *readObject()* som returnerer et Object-objekt.

Kombinasjonen lese/skrive

Vær oppmerksom på følgende:

- ✓ Hvis du bruker *BufferedReader/BufferedWriter*, må du legge til linjeskift (noen maskinmiljøer godtar ikke bare "\n" og da må du kalle *newLine*). Husk at disse er

[32] Klasser som du lager selv, må implementere grensesnittet *Serializable* – et tomt grensesnitt som ikke stiller noen krav til overstyring av metoder.

[33] Forskjellen på et *tegn* (*char*) og en *byte* er at førstnevnte er to bytes i Java og de tolkes utfra et gitt tegnsett. Bytes som ikke kan tolkes (utenfor tegnsettet) "oversettes" til et eget tegn, ofte �.

bufret, så du skriver til et buffer. For å sende bufferet, må du derfor legge til *flush* som tømmer bufferet.

- ✓ *InputStreamReader/OutputStreamWriter* er lite nyttige, fordi de kun leser/skriver tegn (*char*).
- ✓ *DataInputStream/DataOutputStream* er ikke bufret og virker helt greit hos meg.
- ✓ *ObjectInputStream/ObjectOutputStream* er bufret og krever *flush*. De krever imidlertid *ikke* linjeskift. Siden de kan lese/skrive det aller meste – inkludert objekter – er de svært nyttige. Hvis du skriver/leser objekter, må klassen være definert begge steder (du kan legge den til som en *jar*-fil i *libraries* der den mangler).

Eksempel

I eksemplet nedenfor bruker jeg *DataInputStream/DataOutputStream*. Jeg lager en svært enkel tjener som lytter på port 2000 (Linux bruker portene 1 til 1024 til egne formål, så man bør bruke portnummer>1024). Porten må naturligvis være åpen i evt. brannmur.

Serveren

```
public static void main(String[] args) throws Exception {
   //Opprett en serversocket
   ServerSocket minServer = new ServerSocket(2000);
   while (true) { //til programmet stoppes
           //Vent på en klient og skap så en socket
           Socket minVenn = minServer.accept();
           //Skap en leser og les en ordre
           DataInputStream leser =
                   new DataInputStream(minVenn.getInputStream());
           String ordre = leser.readUTF();
           //Skap en skriver og skriv et svar
           DataOutputStream skriver =
                   new DataOutputStream(minVenn.getOutputStream());
           skriver.writeUTF("Ordren " + ordre + " er mottatt\n");
   }
}
```

I den "evige" while-løkken skaper vi en *Socket* som skal kommunisere med en klient som har meldt seg. Metoden *accept* vil vente på en forespørsel fra en klient (imens "henger" programmet). Når en klient har meldt seg, skapes en *Socket* og programmet går videre med lesing fra klienten og skriving tilbake til klienten. Legg merke til at leser/skriver knyttes til input/output-strømmer – oppgitt som argumenter. Metoden *writeUTF* legger til informasjon i starten om hvor lang strengen er. Det utnytter *readUTF* til å lese akkurat riktig antall tegn. Noen kontrolltegn virker ikke (visstnok avhengig av OS).

Klienten

Klienten skrives svært likt men vi lager en *JFrame* og lar brukeren skrive en tekst i et tekstfelt *txtOrdre*. Når brukeren klikker en knapp *butSend* skal denne teksten sendes til tjeneren.

```
private void butSendActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        //Opprett kontakt
        Socket minServer = new Socket("localhost",2000);
        //Skap en skriver og skriv
        DataOutputStream skriver =
                new DataOutputStream(minServer.getOutputStream());
        skriver.writeUTF(txtOrdre.getText() + "\n");
        //Skap en leser og les
        DataInputStream leser =
                new DataInputStream(minServer.getInputStream());
        String lest = leser.readUTF();
        //Vis svaret
        JOptionPane.showMessageDialog(null, lest);
    }
    catch (Exception ex) {
        //TODO: Håndtere feil
    }
}
```

Jeg anser denne koden for å være omtrent selvforklarende.

Ekstra: Serialisering av objekter

Note: Dette er repetisjon fra underliggende kurs og er tatt med for komplettheets skyld.

Objekter ligger i RAM i binær form. Nesten alle objekter refererer til andre objekter. Vi *tenker* gjerne på f.eks. et Person-objekt som et objekt som inneholder et fødselsnummer, et navn osv. I *virkeligheten* inneholder objektet mest referanser.

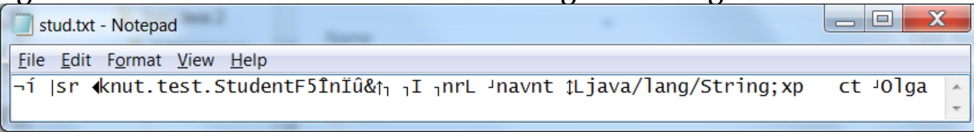
Når et objekt skal sendes over nettet, lagres i en fil e.l. på en måte som gjør det mulig senere å gjenopprette det, må det *serialiseres*. Man må sørge for at ikke bare *referansene* kommer med, men *verdiene*. Serialisering innebærer således *å gjøre om objektet til en strøm med bytes som kan sendes eller lagres og som inneholder alt som trengs for å gjenopprette objektet senere*. Objektets metoder følger aldri med og derfor må klassedefinisjonen være tilgjengelig når objektet skal gjenopprettes.

Man kan lage sitt eget format og selv lage metoden som genererer bytestrømmen, men da må man også selv lage metodene som gjenoppretter objektet ved lesing av strømmen. Det er tungvint. I praksis bruker man derfor ferdige metoder og standardiserte formater, f.eks. XML eller JSON, evt. en seriell, kommalimitert strøm (CSV).

Hvis man vil benytte Javas standard metoder for serialisering – og ikke skrive sine egne – må objektet implementere grensesnittet *Serializable*. Dette er et tomt grensesnitt som ikke krever overstyring av noen metoder. Kompilatoren finner selv ut hvordan serialiseringen bør gjennomføres. For XML, JSON osv. finnes det egne, ferdige metoder i biblioteker.

Det er allikevel tillatt å skrive egne metoder for serialisering, men da må klassen implementere grensesnittet *Externalizable* met metodene *readExternal* og *writeExternal*. Det er ikke vanlig da de automatiske mekanismene for *Serializable* virker utmerket unntatt i svært spesielle tilfeller.

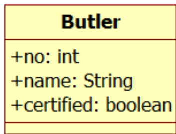
De automatiske metodene for serialisering gir strømmer som ikke er lesbare for mennesker. Her har jeg f.eks. serialisert en student med nr=99 og navn="Olga" til en fil:



Det er jo ikke lesbart bortsett fra noen få tegn her og der. Hvis objektet skal gjenopprettes av en metode som kjenner formatet, spiller jo det ingen rolle.

Oppgave til kapittel 9

Det skal lages et system som registrerer butlere ("James", "Reeves", "Mr. Carson" etc.). I denne omgang skal de ikke lagres, bare holdes i en *TreeMap* slik at de alltid kan vises sortert etter nummer. Alle endringer skal også logges på en server. Bruk *ObjectInputStream* og *ObjectOutputStream*.

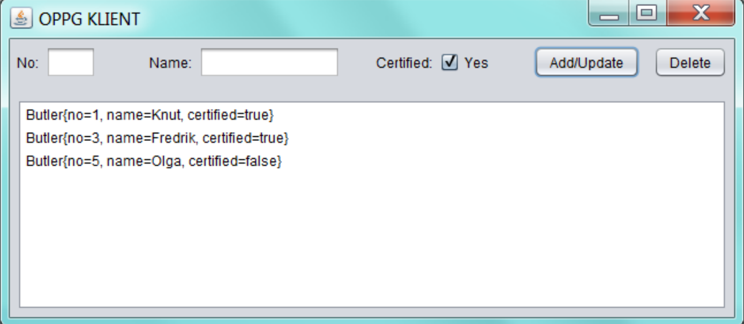


For enkelhets skyld er alle feltene (attributtene) denne gang public og direkte tilgjengelige – selv om det hverken er særlig "pent" eller "trygt".

Det har ingen hensikt å lage kontrollklasser her – algoritmene blir svært korte/enkle og kan godt gjentas hvis det blir flere grensesnitt.

Klienten virker slik

Brukeren fyller ut opplysningene i et skjema og alle registrerte butlere vises i en *JList*. Skjemaet kan se slik ut:

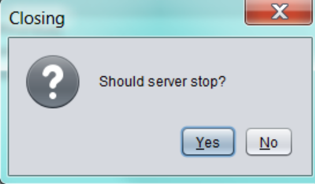


Hvis brukeren klikker i listen, skal feltene vise data for den butleren som ble valgt. Ved å endre opplysningene "name" og "certified", kan brukeren oppdatere. Husk at når du lagrer et objekt som finnes fra før i en *TreeMap*, så blir den eksisterende overskrevet. Å legge til eller endre blir altså samme metode.

Listen skal oppdateres og feltene tømmes hver gang trestrukturen endres. Du kan endre innholdet i listen enkelt med listens metode *setListData* som tar en *Vector* som argument (men en array virker også fint). *TreeMap* har metoden *values* som returnerer alle verdiene ferdig sortert, og de kan enkelt konverteres til en array med *toArray*.

Som du ser har jeg brukt *NeatBeans* forslag til *toString* for butlerne.

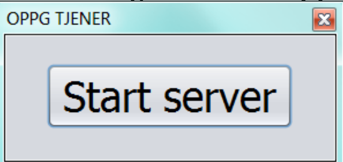
Når klienten lukkes (bruk *closing*-hendelsen) skal brukeren spørres om tjeneren skal stoppes

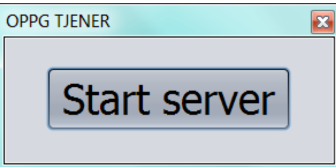


Evt. skal "STOP"-melding sendes til serveren og svaret vises før klienten avslutter.

Serveren virker slik

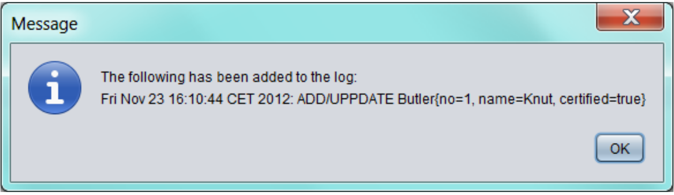
Serveren kan se slik ut før og etter at knappen "Start server" er klikket.





Skjemaet er satt til å ligge over alle andre vinduer hele tiden. Prøv å unngå at det ligger oppå klienten (= ikke vis begge i posisjon 0,0).

Serveren tar imot meldinger om endringene, legger dem til filen "loggfil.txt" med dato og klokkeslett. Serveren bekrefter alle meldinger til klienten som viser bekreftelsen i en meldingsboks.



Serveren har lagt til "The following...", dato og tid, samt returnerer den meldingen den fikk. Alle meldinger til/fra serveren går som tekst (ingen objekter sendes, hvis du da ikke har lyst til å prøve å sende String-objekter).

Hvis serveren får meldingen "STOP" skal den stanse og lukkes. Bruk en Boolsk variabel i while-løkken, f.eks. *while (! avslutt)*, og sett variabelen til *true* når serveren skal stoppe. Da skal den lukke seg med *this.dispose()*. Alle meldinger skal logges, også "STOP", og alle skal gi svar som forklart ovenfor.

Loggen kan se slik ut når en del endringer gjort:

```
Fri Nov 23 15:57:10 CET 2012: ADD/UPDATE Butler{no=1, name=Knut,
certified=true}
Fri Nov 23 15:57:15 CET 2012: ADD/UPDATE Butler{no=5, name=Olga,
certified=false}
Fri Nov 23 15:57:23 CET 2012: ADD/UPDATE Butler{no=3, name=Fredrik,
certified=true}
Fri Nov 23 15:57:31 CET 2012: ADD/UPDATE Butler{no=7, name=James,
certified=false}
Fri Nov 23 15:58:05 CET 2012: DEL Butler{no=7, name=James,
certified=false}
Fri Nov 23 15:57:46 CET 2012: STOP
```

Jeg anbefaler

Lag klientens grensesnitt ferdig uten logging. Bruk en prosedyre for oppdatering av skjemaet. Deretter lager du serveren og legger til logging i klienten i oppdateringsmetoden.

Regn med en del prøving og feiling!

Kapittel 10 – Testing

Generelt om testing

Å teste kode har alltid vært en sentral aktivitet i programmering. Det er derfor viktig at dere vet noe om testing generelt og kan gjennomføre testing i praksis på en systematisk måte.

Også dere studenter tester koden deres, finner feil og retter dem. I en typisk programmeringsoppgave/case, koder dere studenter gjerne entitetsklassene og kontrollklassen først. De testes lite. Så lager dere en GUI og først da begynner den egentlige testingen. Den foregår gjerne ved at dere taster inn verdier i felt, gjør valg osv. og så klikker dere en knapp. Hvis det ikke går bra, leter dere frem hvorfor det feilet (eller ga galt svar), retter og prøver igjen. Denne måten å teste på er svært ustrukturert. Det er på ingen måte sikkert at testingen kommer igjennom alle "veier" i koden. Videre er det et problem at dere regner med at testet kode er korrekt, og tenker sjelden på at senere rettelser kan innføre nye feil i allerede testet kode. Endelig får dere ingen dokumentasjon av *hva* dere har testet.

I profesjonell sammenheng er slik ustrukturert og udokumentert testing ikke på noen måte tilstrekkelig. Da kreves både struktur og dokumentasjon. Godt dokumentert testing kan bidra til forståelse av koden. Dessuten bør det være mulig å gjøre endringer flere år etterpå – av en annen programmerer – og så kjøre alle testene på nytt for å kontrollere at det ikke ble innført nye feil eller at den nye koden "ødelegger" for den gamle. Testingen bør følgelig dokumenteres i elektronisk form.

Det finnes et utall programmer som kan hjelpe til med god testing. Her skal vi bare se på ett av dem som er enkel å bruke fordi den finnes innebygd i NetBeans.

Typer av tester

Man kan skille mellom statisk og dynamisk testing. **Statisk testing** foregår med teknikker som review, walk-through, inspection, Programming Pair o.l. og foregår ved at programmereren selv eller helst andre vurderer koden. **Dynamisk testing** foregår ved at man lager test cases, dvs. data og kombinasjoner av data som man kjører. Ofte utelates den statiske testingen, men dynamisk testing er svært vanlig. Den statiske testingen sjekker at koden kan *verifiseres* (den er korrekt) mens dynamisk testing sjekker at den er *valid* (den produserer riktig resultat).

Den enkleste testen er en **"unit test"**. Da testes hver del for seg, f.eks. en klasse. Hvis alle delene er valide, virker de også som regel korrekt når de samarbeider innen ett program. Unit test er derfor en grunnleggende og sentral del av testingen. Det er dessuten den enkleste fordi man forholder seg til relativt lite kode av gangen. Vanlig er f.eks. å teste hver klasse for seg.

Når unit testen er gjennomført, kan det være aktuelt med **integrasjonstest**. Særlig er det viktig å sjekke samarbeidet mellom to selvstendige programmer, f.eks. vårt program og et databaseprogram, eller vårt program og operativsystemet. Da er grensesnittet mellom dem i fokus (de antas først å være validert hver for seg).

Det er også aktuelt med en **systemtest**, der hele systemet testes mot kravspesifikasjonen for å sikre at totaliteten virker og at f.eks. ikke en del ødelegger for en annen.

Endelig kan man til slutt gjennomføre **akseptansetester** med brukerne. De skal da si seg fornøyd.

Her skal vi konsentrere oss om unit testing.

Regresjonstesting

Regresjonstesting innebærer å kontrollere om det har oppstått nye feil etter en endring, eller kanskje gamle feil har dukket opp igjen. Det er ganske vanlig. Wikipedia sier det slik[34]:

> *Experience has shown that as software is fixed, emergence of new and/or reemergence of old faults is quite common. Sometimes reemergence occurs because a fix gets lost through poor revision control practices (or simple human error in revision control). Often, a fix for a problem will be "fragile" in that it fixes the problem in the narrow case where it was first observed but not in more general cases which may arise over the lifetime of the software. Frequently, a fix for a problem in one area inadvertently causes a software bug in another area. Finally, it may happen that, when some feature is redesigned, some of the same mistakes that were made in the original implementation of the feature are made in the redesign.*

Man kan kontrollere om slike problemer har oppstått ved å kjøre alle gamle tester (som ga aksept tidligere) på nytt. For å få det til, er det helt avgjørende at de opprinnelige testene foreligger i elektronisk form så de lett kan kjøres på nytt. Det er slik vi skal bruke det her.

Det finnes mange andre typer av tester som kan være aktuelle – se den siterte artikkelen fra Wikipedia for en enkel gjennomgang.

Om TTD

Test Driven Design – TDD – er en teknikk der man tester med en gang man har skrevet noe kode. Feil rettes og alle testene kjøres igjen. Teknikken er opprinnelig tett knyttet til Agile Development, men brukes også ellers, uavhengig av utviklingsmetode.

Det er blitt sagt at TDD *ikke* er en måte å finne feil på. TDD beskrives derimot som en designprosess – en robust måte å designe programkomponenter på interaktivt, slik at deres oppførsel spesifiseres ved enhetstesten. Enhetstesten blir altså *spesifikasjonen* for enheten. Den *dokumenterer* hvordan enheten skal virke. Skal man finne feil, er det bedre å bruke automatiserte integrasjonstester.

Teknikken er også blitt kalt "test a little, code a little" og det beskriver den godt. Tanken er å gjennomføre mange små, hurtige iterasjoner:

1. Legg til en ny test
2. Kjør testen for å sikre at den nye testen feiler (det gir en pekepinn om at testen er riktig)
3. Skriv minimum av kode så den nye testen ikke feiler
4. Rett opp koden etter standarder, optimalisering osv. og test igjen
5. Gjenta fra start

I praksis kan vi leve med å lage mer enn bare én ny test og vi kan ta høyde for å unngå feil før vi lager testene. Dette er altså en tilpasning av TDD.

Om JUnit

Vi skal se på *JUnit* – et program for unit testing. Det er innebygget i NetBeans hvilket er en stor fordel. Ellers kan man kjøre JUnit også alene og det finnes en lang rekke alternativer. Flere av dem kan importeres til NetBeans som plug-ins. Vi bruker altså JUnit bare fordi det er praktisk for oss og viser prinsippene godt.

[34] https://en.wikipedia.org/wiki/Software_testing

Slik gjør vi

Vi har laget følgende (altfor enkle) klasse *Emne*:

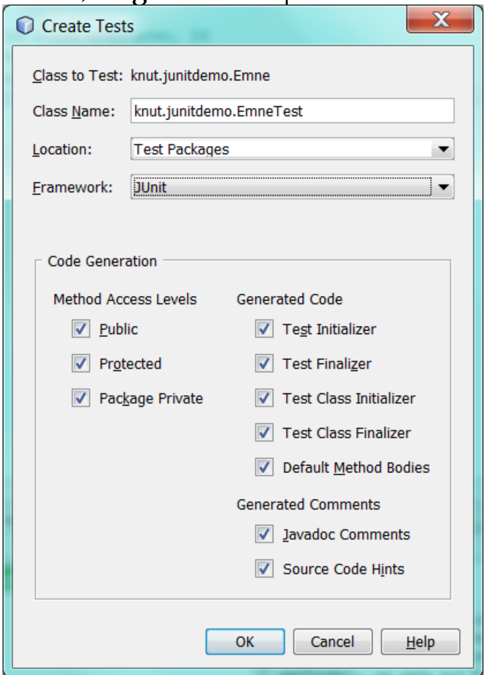
```
public class Emne {
    private String kode; //tre store bokstaver, ID
    private int nr; //100..599, ID
    private float stp; //minst 0
    //avledet: int trinn som skal være første siffer i nr
}
```

Vi vet jo da at det trengs diverse tilgangsmetoder, *toString*, *equals*, *hashCode*, konstruktør og kanskje *compareTo*. Videre skal vi lage den avlede metoden *getTrinn*.

Bak feltdeklarasjonen står det spesifisert hva brukerne krever av verdiene. Vi må sikre at koden kontrollerer at disse beskrankningene – også kalt "business rules" – overholdes. Det kan se enkelt ut, men faktisk kan selv så enkle eksempler fort bli kodet galt, så vi må teste.

1. Skap testen

Med klassen som aktiv i NetBeans, velger vi Tools|Create Tests:



Vi kan velge mellom *JUnit* og *TestNG*. De er svært like og vi bruker *JUnit*. Vi blir da bedt om å velge mellom versjon 3 og 4 og velger selvsagt den seneste (den forrige er for dem som skal kjøre gamle tester om igjen – versjon 4 har flere muligheter). Det genereres da en testklasse kalt *EmneTest* og det er her vi skal dokumentere og kjøre testene.

Det er allerede laget diverse testmetoder med annotering (jeg har fjernet tomme linjer):

```
public class EmneTest {
    public EmneTest() {
    }
    @BeforeClass
    public static void setUpClass() {
    }
    @AfterClass
    public static void tearDownClass() {
    }
    @Before
    public void setUp() {
    }
    @After
    public void tearDown() {
    }
    @Test
    public void testSomeMethod() {
        // TODO review the generated test code and remove the default
call to fail.
        fail("The test case is a prototype.");
    }
}
```

Linjene som begynner med @ er annoteringer dvs. beskjeder til kompilatoren.

1. *BeforeClass* og *AfterClass* kjøres henholdsvis før testen begynner og helt til slutt.
2. *Before* og *After* kjøres før hver eneste test-metode
3. *Test* angir en testmetode.

Det at noe er en testmetode, innebærer at hvis den feiler så logges det en melding om det, men programmet fortsetter å kjøre (også uten try-catch). De er altså spesielt laget for testing. Det er disse testmetodene som interesserer oss i unit tester – de andre er mer interessante i integrasjonstester og kan fjernes her.

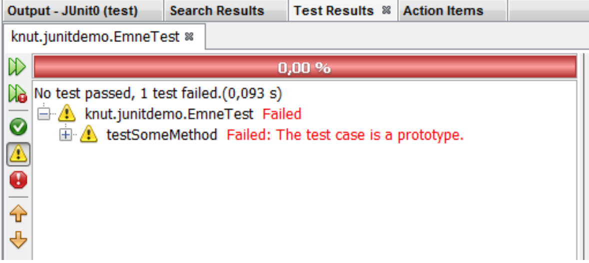
Den ene testmetoden som er laget, *testSomeMethod* er laget for å feile. Setningen

```
fail("The test case is a prototype.");
```

tvinger testen til å feile med den angitte feilmeldingen i loggen.

2. Kjør testen

Vi kan nå kjøre testen. Vi kan ikke kjøre prøvekoden på vanlig vis for vi har ingen *main*-metode. Isteden må vi høyreklikke på testklassen og be om *Run File*. Vi får da følgende logg:



Ikke uventet blir det rapportert at det ble kjørt én test (metoden *testSomeMethod*) og den feilet. Noe er altså feil i klassen *Emne* eller så er kanskje testmetoden feil. Vi må rette koden.

3a. Skriv minimum av kode

Den minste kode for *setStp* og *getStp* (de enkleste her) kan være den som genereres av NetBeans:

```
public float getStp() {
    return stp;
}
public void setStp(float stp) {
    this.stp = stp;
}
```

3b. Test denne koden til den er korrekt

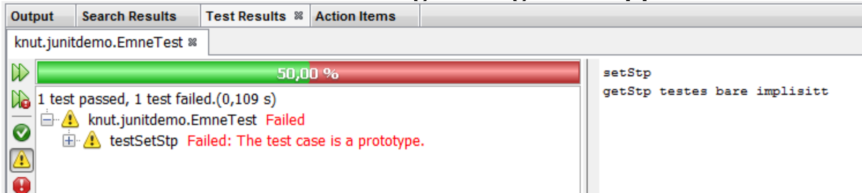
Jeg ber nå pány om å få generert testkode (se pkt 1 ovenfor) og JUnit legger til flere testmetoder:

```
@Test
public void testSomeMethod() {
    // TODO review the generated test code and remove the default call
to fail.
    fail("The test case is a prototype.");
}
/**
 * Test of getStp method, of class Emne.
 */
@Test
public void testGetStp() {
    System.out.println("getStp");
    Emne instance = new Emne();
    float expResult = 0.0F;
    float result = instance.getStp();
    assertEquals(expResult, result, 0.0);
    // TODO review the generated test code and remove the default call
fail.
    fail("The test case is a prototype.");
}
/**
 * Test of setStp method, of class Emne.
 */
@Test
public void testSetStp() {
    System.out.println("setStp");
    float stp = 0.0F;
    Emne instance = new Emne();
    instance.setStp(stp);
    // TODO review the generated test code and remove the default call
to fail.
    fail("The test case is a prototype.");
}
```

Den første *testSomeMethod* er nå uinteressant og vi kan slette den. Når det gjelder testing av *getStp* så er det fint lite den kan gjøre feil, så den tester jeg heller implisitt i andre testmetoder. Jeg retter den følgelig så den ser slik ut:

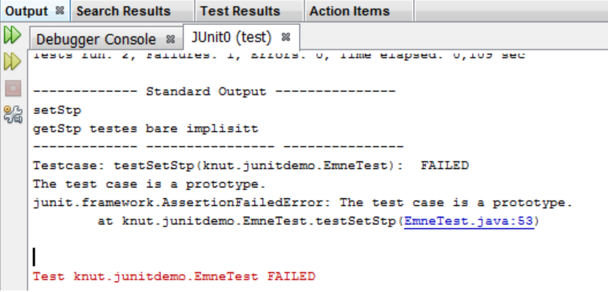
```
@Test
public void testGetStp() {
    System.out.println("getStp testes bare implisitt");
}
```

Nå kjører jeg testen (Run File som ovenfor) og får følgende rapport:



Det er kjørt to testmetoder – *getStp* gikk selvsagt greit igjennom – den skrev bare en melding til konsollet – *setStp* feilet. Vi må altså analysere den litt nærmere.

I Output-vinduet kan vi finne flere opplysninger:



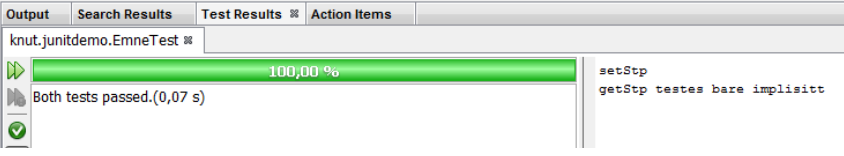
Lenken bringer oss dit i koden hvor feilen oppsto.

I dette tilfellet feiler koden fordi vi uttrykkelig har bedt om det med setningen *fail*. Testen skal ikke feile med argumentet null etter business rules. Testkoden er feil – vi må stryke *fail* og erstatte det med en *assert*-setning:

```
assertEquals("Forventet 0 fikk "+instance.getStp(),stp,
    instance.getStp(),0);
```

assertEquals ("påstå at noe er likt") har først en melding som skal logges hvis noe er galt, deretter hva vi forventer å få, hva vi fikk og maksimal godtatt forskjell på de to (fordi det er en float og avrundinger kan gi bitte små forskjeller som vi aksepterer).

Jeg får beskjed om at alt er i orden:



3c Test flere test-cases

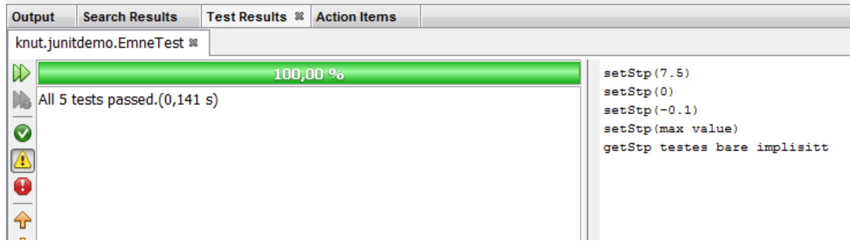
Dette er jo lovende: Testen viser at med argumentet 0 – *men bare da* – så virker koden *setStp* tilsynelatende riktig etter business rules. Vi er imidlertid ikke ferdige med det – vi må lage flere test cases, f.eks. 7.5 (en typisk verdi) og Float.MAX_VALUE.

Generelt bør vi teste laveste, høyeste og minst én midt imellom. Ulovlige verdier bør også være med, gjerne litt under nedre og litt over øvre grense.

Jeg legger nå inn -0.1 som test-case. Jeg kan da kopiere koden for *testSetStp* og rette testverdien, loggen og utskriften samt metodenavnet:

```
        @Test
    public void testSetStp2() {
        System.out.println("setStp(7.5)");
        float stp = 7.5F;
        Emne instance = new Emne();
        instance.setStp(stp);
        assertEquals("Forventet 7.5 fikk "+instance.getStp(),stp,
                instance.getStp(),0);
    }
            @Test
    public void testSetStp3() {
        System.out.println("setStp(max value)");
        float stp = Float.MAX_VALUE;
        Emne instance = new Emne();
        instance.setStp(stp);
        assertEquals("Forventet max value fikk "+instance.getStp(),stp,
                instance.getStp(),0);
    }
        @Test
    public void testSetStp4() {
        System.out.println("setStp(-0.1)");
        float stp = -0.1f;
        Emne instance = new Emne();
        instance.setStp(stp);
        assertEquals("Forventet -0.1 fikk "+instance.getStp(),stp,
                instance.getStp(),0);
    }
```

Vi tester igjen og da testes alle metodene:



4. Rett opp koden

Alle testene passerte og det var ikke bra! Den burde gitt feil for argumentet -0.1. Det indikerer feil i *setStp* som må rettes opp, f.eks.

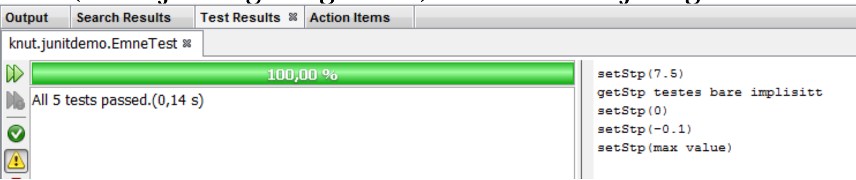
```
    public void setStp(float stp) {
        if(stp<0)
            throw new IllegalArgumentException("Stp må være minst 0");
        this.stp = stp;
        }
```

Testingen gir da én feil som forventet. Vi får ingen loggmelding fra *assert*-setningen fordi den faller ut tidligere med Exception.

Dette ble litt dumt: Testen feiler og vi er fornøyd med det! Det bør vi endre på. Jeg forandrer *testSetStp4* som tester med ulovlig testverdi:

```
@Test(expected = IllegalArgumentException.class)
public void testSetStp4() {
    System.out.println("setStp(-0.1)");
    float stp = -0.1f;
    Emne instance = new Emne();
    instance.setStp(stp);
    fail("Ulovlig verdi ble godtatt: -0.1");
}
```

Jeg har lagt til hva som forventes bak *@Test* og endret *assert* til *fail*. Det første indikerer at testen har suksess hvis *testSetStp4* faktisk feiler, den andre vil tvinge den til å feile hvis den går greit igjennom (det er jo da egentlig en feil). Nå viser testkjøringen at alt er som forventet:



5. Gjenta fra 1

Nå godtar vi *setStp* og er klare for mer kode, f.eks. *setNr* og *getNr*. Arbeidsmåten blir den samme, så jeg viser den ikke her.

Forenkling av testkoden

I eksempelet ovenfor, kopierte jeg kode og rettet den litt. Det er naturligvis ikke bra fordi jeg da replikerer mye kode. Vi bør tenke mer generelt. Vi bør også standardisere navn.

Jeg bruker arrays[35] til test-casene. Array-navnene har fast prefiks

- ✓ prefiks "e" = test-casene forventes å feile (e for exception)
- ✓ prefiks "a" = test-casene forventes å gå greit (a for assert)
- ✓ prefiks "f" = test-casenes forventede resultat (f for forventet)

Alle test-casene samles øverst så det er lett å finne dem og enkelt å kopiere dem alle inn i annen dokumentasjon.

Testmetoden har prefiks "test" etterfulgt av metodens navn samt suffiks "Feiler" og "OK", f.eks. testSetNavnFeiler og testSetNavnOK.

Hver testmetode skriver navnet sitt til *System.out*. Da kan jeg se at metoden faktisk har kjørt. Noen ganger kan det være behov for å skrive ut mer, til og med inne i testløkken.

[35] Her skal ikke innholdet i arrayen endres under kjøring, derfor gjelder ikke de ulempene som er nevnt tidligere (kapittel 4, side 45) så arrays egner seg godt.

```
    /*
     * Testdata
     * prefiks "e" = forventet å feile
     * prefiks "a" = forventet å gå greit
     * prefiks "f" = forventet resultat
     */
    //setStp
    private final float[] eStp = {-1};
    private final float[] aStp = {0, 7.5f, Float.MAX_VALUE};
    private final float[] fStp = aStp;

    /**
     * Test of setStp method, of class Emne.
     */
    @Test(expected = Exception.class)
    public void testSetStpFeiler() {
        Emne instance = null;
        System.out.println("setStpFeiler");
        for (float stp : eStp) {
            instance = new Emne("INF", 100, 7.5f);
            instance.setStp(stp);
        }
    }

    @Test
    public void testSetStpOK() {
        System.out.println("setStpOK");
        Emne instance = null;
        for (float stp : aStp) {
            instance = new Emne("INF", 100, 7.5f);
            instance.setStp(stp);
            assertEquals(stp, instance.getStp(), 0);
        }
    }
```

Som du ser så bygger jeg først opp to float-arrays, den ene med lovlige verdier og den andre med ulovlige. Vanligvis kan det også trenges én med forventede verdier i samme rekkefølge som de lovlige men her forventet det samme som de lovlige verdiene. Videre lager jeg bare to testmetoder, én som forventer at metoden skal feile og én med *assert* som forventer at den ikke skal feile. Der går jeg igjennom alle test-cases i en for-each-løkke. Når denne går igjennom uten at det rapporteres testfeil, så vet jeg at alle de lovlige og ulovlige verdiene som jeg forsøkte ga riktig resultat.

Når assert-setningen må vise både til argumentverdi og resultat, kan vi ikke bruke for-each. Vi må isteden bruke en indeks på test-casene:

```
        @Test
        public void testSetKodeOK() {
            System.out.println("setKodeOK");
            Emne instance = null;
            for (int i = 0; i < aKode.length; i++) {
                instance = new Emne(aKode[i], 100, 7.5f);
                assertEquals(fKode[i], instance.getKode());
            }
        }
```

Jeg bruker *aKode[i]* og resultatet sammenliknes med *fKode[i]* i assert-setningen. Siden de skal sees i sammenheng, må jeg bruke en indeks.

Noen metoder egner seg ikke for testing på denne måten. Det er først og fremst behov for å teste metoder som inneholder en algoritme med programlogikk eller som skal implementere "business rules". De fleste get-metodene er så enkle at de faller utenfor dette behovet. Mange

argumenter for at bare *synlige* metoder skal testes – altså klassens API. Jean-Baptiste Rieu er av dem som mener det i sin sjekkliste (inkludert senere i dette kapitlet, side 101).

Det diskuteres heftig blant fagfolk om man skal teste private metoder. Én skriver f.eks. i Stackoverflow at

> *The best way to test a private method is via another public method. If this cannot be done, then one of the following conditions is true:*
> 1. *The private method is dead code*
> 2. *There is a design smell[36] near the class that you are testing*
> 3. *The method that you are trying to test should not be private*

Andre mener at dette blir for snevert, da private metoder kan trenge mer testing enn tilgjengelig gjennom de offentlige metodene. Mitt eget syn heller til at det holder å teste klassens synlige API fordi private metoder er laget enten for å forenkle (dele opp) en public metode eller fordi flere metoder trenger den (unngå koderedundans). Uansett må den bli kalt av minst én public metode ellers er den "død kode" (pkt. 1 ovenfor) og skal fjernes.

Det kan i alle fall være interessant å dokumentere hvilke metoder man ikke har testet direkte. Ovenfor viste jeg at man kan lage en utskrift for de metodene som ikke testes:

```
@Test
public void testGetStp() {
    System.out.println("getStp testes bare implisitt ved bruk");
}
```

Kan vi stole på testene?

Det kan være lett å tro at når vi har testet metoden *setStp* slik, så vet vi at den er korrekt. Det vet vi ikke! Det vi faktisk vet er bare at (1) den virker greit *med de test-casene* vi har forsøkt og (2) forutsatt at *testkoden er korrekt.*

Generelt kan testing aldri bevise at koden er korrekt. Einstein sa det omtrent slik: "Ingen eksperimenter kan noen gang vise at jeg har rett – når som helst kan et eneste eksperiment vise at jeg tok feil". Det samme gjelder for testing av kode.

Hva om f.eks. argumentet er *null*? Svaret er for så vidt greit her: En *float* er en primitiv datatype som aldri kan være *null* (det er det bare objekter som kan være). Hvordan vet vi om koden er korrekt også for *Float.MIN_VALUE*? Det vet vi ikke – vi må eventuelt prøve.

Noe har sluppet igjennom, brukeren rapporterer en feil

Oops – en ikke ukjent situasjon. Da må man *først* lage en test-case som feiler slik brukeren opplevde. *Deretter* retter vi koden så feilen blir borte. Hele tiden kjører vi *alle testene* så vi er rimelig sikre på at ikke vi har innført nye feil.

Det antydes i faglitteraturen at slike endringer ofte gjøres under tidspress og det er fristende å rette koden så akkurat dette tilfelle går greit, men man bør tenke mer generelt og lage flere case-tests for denne ene feilen. Hvis vi f.eks. ovenfor hadde glemt test-caset stp=-0.1 og følgelig heller ikke fått med testen `if(stp<0){......}`. Så ble det rapportert at programmet feilet når stp var -3. Da må vi tenke lenger enn til bare å legge inn `if (stp == -3.0f) {......}` for feilen er mer generell enn som så.

En feilrapport må alltid sees som et symptom på en generell feil.

[36] "Design smell" er et slanguttrykk for at designet er dårlig.

Et annet problem er når metoder har mer enn ett parameter. Da må vi nøste iterasjonene så vi får prøvde alle kombinasjoner av test-cases. Slik er det f.eks. med konstruktøren. Her tester jeg alle kombinasjoner som skal gå greit:

```
@Test
public void testEmneOK() {
    System.out.println("testEmneOK");
    Emne instance = null;
    for (int i = 0;i<aKode.length;i++) {
        for (int nr : aNr) {
            for (float stp : aStp) {
                instance = new Emne(aKode[i], nr, stp);
                assertTrue(fKode[i].equals(instance.getKode())
                        && instance.getNr()==nr &&
                      instance.getStp()==stp);
            }
        }
    }
}
```

Legg spesielt merke til assert-setningen her. Jeg bruker indeksert *aKode* som argument til konstruktøren og sammenlikner indeksert *fKode* med objektets *getKode*.

Videre lager jeg en *exception-test* der jeg prøver konstruktøren med alle kombinasjoner av ulovlige argumentverdier. Egentlig burde jeg nok også tatt med kombinasjoner med bare én ulovlig og to ulovlige. Denne testen av konstruktøren kan sees på som et anslag til integrasjonstest.

Litt triksing

Selv synes jeg det blir litt for omstendelig å teste litt, kode litt osv. Jeg tar for sjansen på en litt annen arbeidsmåte: Jeg skriver alle metodene først, helt uten kontroller. Så lager jeg testklassen og sjekker at den feiler 100 %. Deretter lager jeg test-cases utfra business rules (og annet). Så korrigerer jeg testmetodene så de prøver det som skal feile og det som ikke skal feile (en testmetode med forventet feil og en med assert). Normalt vil begge feile 100 %. Så retter jeg kode i én metode av gangen til begge dens tester går greit igjennom, før jeg begynner på neste metode osv.

Som test-cases tenker jeg

1. *null* for alle objekter
2. en nedre grenseverdi, øvre grenseverdi og en typisk verdi eller flere
3. verdi under grensen og én over grensen
4. norske tegn i strenger
5. små kontra store bokstaver i strenger

De fleste feil jeg gjør fanges faktisk av kompilatoren. Det er irriterende, men svært, svært nyttig.

Ellers er det en god tommelfingerregel at hvis koden er lett å forstå for andre, så er sannsynligheten størst for at den er riktig. Skriv altså klar, ryddig kode! Bruk NetBeans mulighet for å be om *Format* med høyreklikk.

Noen systemer (ikke JUnit) kan generere et stort antall test-cases tilfeldig etter visse regler og kjører samtlige. Problemet da er gjerne å se – på lange utskrifter – at den forventede verdien er korrekt. En bedre metode kan være å hente "levende data" fra et annet system og kjøre dem men problemet med riktig resultat er det samme.

Er det verd det?

Du tenker kanskje nå at dette er mye arbeid? Ja, det er mer arbeid enn å taste inn noen prøvedata i en GUI og håpe på det beste. Hva er det da man oppnår med denne ekstra innsatsen?

1. Kvaliteten av programmet øker betydelig. Systematisk, dokumentert testing er veldig mye bedre enn ad hoc, manuell, ustrukturert testing.
2. Vi finner feil i små, avgrensede moduler mens de ennå er enkle å finne/rette. Til syvende og sist kan det spare tid som ellers sløses bort på komplisert feilfinning i en stor kode.
3. Vi får automatisk dokumentert hva vi har testet som test-cases. Det gjør det enklere å forstå hva forrige programmerer tenkte under kodingen og hvordan vedkommende forsto "business rules".
4. Vi får et klart skille mellom feil i en kodemodul som vi retter straks, og feil som oppstår når modulene samarbeider i programmet (og nye feil oppstår). Vi kan rimelig trygt anta at feilene oppsto i samarbeidet, ikke i den enkelte modul. Med "tast i GUI-metoden" kan du ikke ane hva slags feil det er.
5. Testing blir regressiv så alle tester kjøres hver gang. Derved sjekker man om endringer har innført nye feil (eller gamle feil dukker opp igjen). Antallet test-cases øker hele tiden.
6. Det er faktisk moro og det gir mye kodetrening. Det har vi alle godt av.

Ekstra: Teste private metoder

Som nevnt ovenfor (side 98) diskuteres det intenst om private metoder skal testes i enhetstesting. Bill Venners gir en fin oversikt over denne diskusjonen og hvordan det eventuelt kan gjøres på http://www.artima.com/suiterunner/privateP.html. Kort fortalt diskuteres han følgende muligheter:

1. *Ikke test private metoder.*
   På den ene side blir slike metoder ofte skrevet bare for å gjøre én annen metode enklere å forstå. Da blir den testet implisitt når den andre metoden testes. På den annen side vil man være sikrere på at all koden er korrekt hvis den testes. Derfor kan det være fristende.
2. *Gi den private metoden pakketilgang.*
   Det gjør du ved ikke å sette på synlighet (synlighet er oppsummert i et eget underkapittel på side 130). Så lenge testmetodene da er i samme pakke som klassen som skal testes, vil metoden være synlig. Testene kan lages som ellers. Dette krever at testklassen ligger i samme pakke som den testede klassen. Jeg har generelt anbefalt at testklassene legges i egen mappe, men i samme pakke som det som skal testes, så da vil denne strategien virke fint. Ulempen er jo at alle andre klasser i pakken også gis tilgang.
3. *Lag testklassen som en indre (nøstet) klasse i klassen som skal testes.*
   Indre klasser har tilgang til private medlemmer i den klassen de er en del av. Ulempen er at testklassen ender som en del av den ferdige, kompilerte bytekoden og gjør den større. Det kan også være vanskelig å få kjørt testen.
4. *Bruk refleksjon.*
   Refleksjon innebærer muligheten for å se og endre private medlemmer under kjøring (innenfor sikkerhetsbegrensninger). Wikipedia[37] forklarer det slik: "reflection allows *inspection* of classes, interfaces, fields and methods at runtime without knowing the names of the interfaces, fields, methods at compile time. It also allows *instantiation* of new objects and *invocation* of methods." Poenget er jo nettopp at man fra utsiden ikke kan "se" private medlemmer, men refleksjon gjør det altså allikevel mulig å inspisere dem. Til dette brukes grensesnittet java.lang.reflect. Det er mange som argumenterer heftig for at dette er den beste metoden. Fordelen er at de private metodene blir testet, samtidig som man hverken endrer synligheten til

[37] http://en.wikipedia.org/wiki/Reflection_%28computer_programming%29

"pakkesynlighet" eller bruker indre klasser. Det er imidlertid ganske komplisert og ikke særlig enkelt for nybegynnere. I Visual Studios testmiljø er det slik det gjøres.

Ekstra: Unit Testing Checklist

av *Jean-Baptiste Rieu* - http://java.dzone.com/articles/unit-testing-checklist

A **unit test** is a set of methods launched frequently to validate your code. It is usually a good idea to write code in this order:

1. Write a class API
2. Write a method to test the API
3. Implement the API
4. Launch the unit tests

Why write unit tests? They validate current and future implementations. They measure code quality. They force you to write testable, loosely coupled code. They're cheaper than manual regression testing. They build confidence in your code. They help teamwork.

Why use a checklist? Unit testing can be harder than actual implementation. Unit testing forces you to really think things through. But unit tests should be simple, direct, and easy to read and maintain. You also need to know when to stop writing tests and start writing the implementation.

Use this checklist to be sure your tests are really useful and to the point.
Remember: the checklist helps you avoid big mistakes, but you need to make sure of the following:

ICON KEY: ? reason why ! attention ex example i further information

O My test class is testing only one class.
? *You are testing a class API to be sure the public contract is respected.*
O My methods are testing only one method at a time.
! *Be sure not to test private methods! They are hidden implementation details, not API.*
O My variables and method names are explicit.
ex *For example, store an expected value in an expectedFoo variable instead of just foo. If you test many combinations, use composed variable names like inputValue_NotNull, inputValue_ZeroData, inputValue_PastDate, etc. (according to your application's coding convention).*
O My test cases are easy to read by humans.
? *Future maintainers should be able to read your tests before reading the implementation. This will help them understand a class API before tweaking or debugging it.*
O My tests respect the usual clean code standards.
i *There should be no flow control in a test method (switch, if, etc.). A good test is just a very straightforward sequence of setup/validate instructions. If necessary, use sub-methods to factorize and make your tests easier to read. In case of multiple scenarios, use multiple test methods (one for each case).*
ex *For example, a test method should fit on screen without scrolling – 1 to 20 lines long. If the method is longer, consider writing multiple test methods for each case instead of jamming them together.*
O My tests are also testing expected exceptions.
ex *In java, use* **@Test(expected=MyException.class)**.
O My tests don't need access to a database.
! *Or if a test does need database access, then it must be a mocked, "fire and forget" temporary database that you fill with test cases for* **every new test method** *(use the Setup/Teardown methods to prepare it).*
O My tests don't need access to network resources.

You can't rely on third parties like network or device presence to validate a method (use mocks).
O My tests control side effects, limit values (max, min) and null variables (even when they throw exceptions).
You want to make sure these problem cases never occur, even when the test won't be used during maintenance
O My tests can be run at any time, at any place without needing configuration or human intervention.
O My tests pass for the current implementation **and** are easy to evolve.
Tests really exist to support code evolution. If they are too hard to maintain or too light to refine the code, then they are a useless burden. (Many developers avoid unit testing for this reason.)
O My tests are concrete.
In Java: don't use **Date()** *as input for a method you are testing, but build a concrete date out of* **Calendar** *(don't forget to force the timezone). Other example: use* **name = "Smith"***; instead of* **name = "name"***; or* **name = "test"***;*
O My tests use a mock to simulate/stub complex class structure or methods.
Remember to test only one class API at a time.
Never test third-party libraries through your own classes. Libraries should come with their own tests (this is actually a good way to choose a library).
O My tests are never @ignored or commented out. ***Never. Ever.***
O My tests help me validate my architecture.
If you can't test a method or a class, your design is not agile enough.
O My tests can run on any supported platform, not just the targeted platform.
Don't expect a particular device or hardware configuration. Otherwise, your tests will make migration tougher and you will be incentivized to disable them.
O My tests are lightning fast!
Slow tests shouldn't drag you down. Speed encourages you to launch your tests often. It also helps to reduce building time on Continuous Integration systems.
Use a test runner that allows you to launch one test at a time while you are writing it. Use "delay" or "sleep" with caution – i.e., only in some edge cases, like waiting for notifications or clock-based methods.

Oppgave til kapittel 10

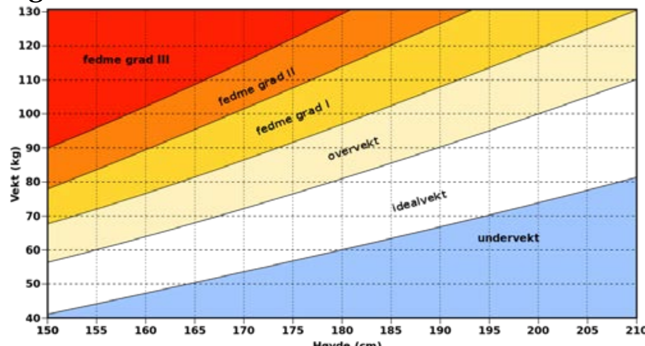
I en screeningundersøkelse skal et stort antall voksne (over 18 år) trekkes ut tilfeldig. De skal måles og veies, og man beregner BMI (Body Mass Index, på norsk KMI Kroppsmasseindeks). Man vil på denne måten finne ut hvordan BMI fordeler seg i befolkningen, etter kjønn og alder. Undersøkelsen skal være anonym, og derfor bruker man et avledet *regnr* som ID.

BMI beregnes ved å ta vekten (i kg) og dele på høyden (i meter) kvadrert:

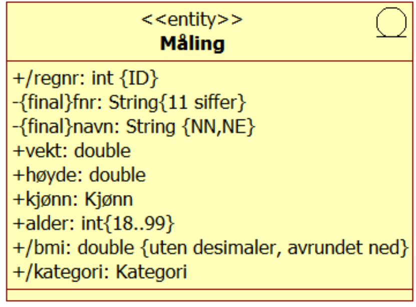
$$BMI = \frac{vekt}{høyde^2}$$

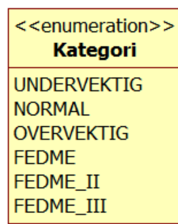
Ut fra BMI gir man en karakteristikk som følger[38]:

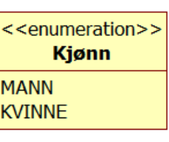
Kategori	BMI [kg/m²]
Undervektig	< 18,5
Normal	fra 18,5 til 24,9
Overvektig	fra 25 til 29,9
Fedme	fra 30 til 34,9
Fedme II	fra 35 til 39,9
Fedme III	> 40



For å få registrert dette, skal det lages én klasse og to enumerations:







Note: UML-diagrammet er tegnet etter vår "standard" fra INF251 OOAD. Da er attributter merket / avledet dvs de returnerer en *beregnet* verdi med get-metoden. Videre skal attributter merket private hverken ha public get- eller public set-metode. NN=not null og NE=not empty string.

Om attributtene får vi opplyst:

- ✓ *regnr* er objektets *hashcode*.
- ✓ *FNR* skal være akkurat 11 siffer. Du behøver *ikke* sjekke at det faktisk er et lovlig fødselsnummer[39].
- ✓ *NAVN* skal ikke være null og ikke tom streng.
- ✓ *vekt* er angitt i kg med fritt antall desimaler (f.eks. 72,3 kg).
- ✓ *høyde* er angitt i meter med fritt antall desimaler (f.eks. 1,78).
- ✓ *bmi* er beregnet etter formelen ovenfor avrundet ned til nærmeste hele tall (f.eks. 22,0).
- ✓ kategori er basert på *bmi* etter tabellen ovenfor (f.eks. *Kategori.NORMAL*).
- ✓ *toString()* skal gjengi alle data unntatt *FNR* og *NAVN* pga anonymiteten.
- ✓ To objekter er like hvis de har samme *FNR*.

Du skal nå gjennomføre Test Driven Design (nesten) "etter boka". Du behøver ikke å teste private metoder.

Oppgave A

Lag klassen helt uten kontroller (dvs. feltet tilordnes uten noen kontroll av verdien). Lag også begge enumerasjonsobjektene, men dem behøver du ikke teste.

Oppgave B

Lag deretter test-cases komplett for hver metode.

---

[38] Wikipedia, https://no.wikipedia.org/wiki/BMI noe tilpasset.

[39] Se evt. j00rn på http://www.diskusjon.no/index.php?showtopic=738727

Oppgave C
Kjør full unit-test og se etter at *alle testene feiler*.

Oppgave D
Korriger koden litt etter litt og test igjen og igjen helt til *ingen av testen feiler*.

Kapittel 11 – Java-bibliotek

Hva er et bibliotek

En kilde[40] definerer biblioteker slik:

> *"A library is a reusable software component that saves developers time by providing access to the code that performs a programming task".*

Enkelt sagt dreier det seg om en samling klasser beregnet på bruk i andres applikasjoner. De har følgelig ingen klasse med *main*-metode. Klassene kompileres til bytekode og brukes ferdig kompilert. Det gjør at programmereren som skal bruke dem, er helt avhengig av god dokumentasjon (JavaDoc) for alt som er synlig. Det er viktig at klassene er programmert med tanke på gjenbruk. Grensesnittet (API) bør inkludere bare det som neste programmerer behøver å se – resten skjules. Man tenker også nøye gjennom hvilke navn klasser og medlemmer bør ha og gjør det fleksibelt. I Java finner du f.eks. *LinkedList<E>* som gjør det mulig å lage lister for alle typer objekter med streng typekontroll.

Pass på at du ikke forveksler *bibliotek* med *pakke* (*package*). Et **bibliotek** har en samling av klasser som andre programmerere kan benytte. Implementeringen er gjerne innkapslet (skjult) så programmereren vil benytte klassenes API som er beskrevet i medfølgende dokumentasjon.

En **pakke** brukes primært som et navnerom. Da er det helt i orden å ha to, synlige navn/metoder som er helt like hvis de bare ligger i hver sin pakke (da kvalifiseres navnet ved å angi pakkenavnet foran med punktum etter). Pakker er også av betydning for synlighet. Avhengig av tilgangsdeklarasjonene (*access modifiers*, dvs. *private*, *protected* osv.) er klasser og medlemmer synlige bare innenfor eller også utenfor pakken.

Lage biblioteket

Når du skal lage et bibliotek med NetBeans, skaper du et nytt prosjekt. Du kan be om en ny *Java Library* eller en ny *Java Application* men da **uten main-class** (det er jo slik vi pleier å gjøre det uansett).

Du lager klassene akkurat som vanlig, men i tillegg må du

- ✓ skrive god dokumentasjon inkludert *package-info* og husk å sette *Browser Window Title*
- ✓ generere JavaDoc
- ✓ be om *Build* (*Run*-menyen)
- ✓ prøve biblioteket i et annet program så du ser at det virker som forutsatt

Du må imidlertid ikke:

- ✓ *ikke* lage *main*-metoden
- ✓ *ikke* "prøvekjøre" biblioteket – uten *main* lar det seg ikke kjøre men det bygger filer

Når biblioteket er ferdig, vil mappen *dist* (for *distribution*) inneholde en jar-fil med alle klassene pakket i ett. Det er det biblioteket du kan distribuere og/eller bruke selv i et annet program. Det er slike vi har hentet ned tidligere bl.a. for å kjøre mot databaser.

Bruke biblioteket

Jeg foretrekker å kopiere bibliotekene selv til en egen mappe i mitt prosjekt som jeg kaller *lib*.

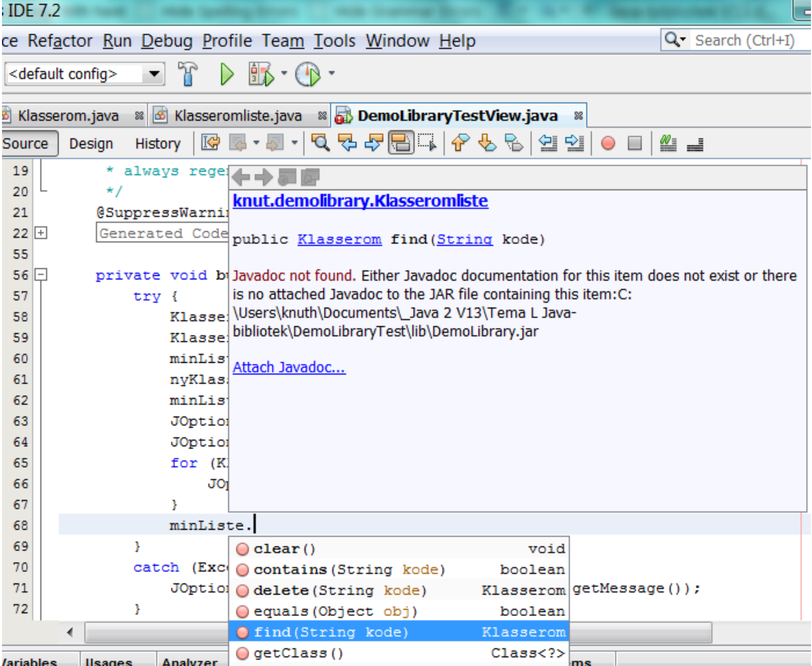
I prosjektet, under *Libraries* legger jeg til en referanse til jar-filen med biblioteket. Da blir alle klassene i biblioteket tilgjengelig i mitt program, og jeg får den vanlige hjelpen av NetBeans når jeg skriver kode.

---

[40] http://www.digilife.be/quickreferences/PT/Build%20your%20own%20Java%20library.pdf

Husk at hvis du finner feil når du tester biblioteket og endrer det, så får det ingen virkning for testprogrammet før du har slettet referansen til jar-filen og referert til den nye versjonen.

PS
Hvis du ikke får opp dokumentasjonen når du bruker biblioteket, men får denne meldingen:



klikker du lenken "*Attach Javadoc* "og legger til dokumentasjonen der. Kopier gjerne mappen "*javadoc*" fra submappen "*dist*" inn i ditt eget prosjekt, og henvis til denne *mappen*.

Det er det hele. Prøv det selv ved å løse ukeoppgaven.

Demo: Klasserom

Jeg har laget et Java-bibliotek med to klasser: *Klasserom* og *Klasseromliste* på den måten som er forklart ovenfor. Du kan selv se på koden hvordan det er gjort.

Oppgave til kapittel 11

I tillegg til å øve på å lage et bibliotek, repeterer denne oppgaven tegning av klassediagram, å lage klasser med NetBeans og å dokumentere med JavaDoc. Alt nyttig for eksamen.

Ved HBV skal det lages en rekke nye, administrative systemer. Ett av dem skal holde orden på kurs. Kurs defineres som "gjennomføring av undervisningen for ett emne" (som mange kaller "fag"). Tiden for kurset angis med eksamenssemester bestående av årstall etterfulgt av "V" for vår eller "H" for høst. f.eks. 2014H, 2015V. Hvert kurs har en fagansatt som er ansvarlig. Det er bare én ansvarlig, selv om flere underviser deler på undervisningen. Hvert kurs gjelder bare ett emne.

Man vil ha et bibliotek for de sentrale entitetsklassene i denne forbindelse.

1. Lag først klassediagram for biblioteket med *StarUML*. Husk å gjøre det så fleksibelt som mulig.

2. Lag deretter biblioteket. Du behøver ikke legge inn så mange felt i hver klasse. Dokumenter med *JavaDoc* (kapittel 3). Prøvekjør *NetBeans* funksjon *Format* (høyreklikk i koden).

3. Prøve biblioteket med et enkelt testprogram som du lager, eller be helst en studentkollega prøve biblioteket ditt.

Kapittel 12 – Tråder i Java

Hva er tråder (threads)?

Et *program* er en ferdig applikasjon som kan kjøres. Når du starter/åpner et slikt program, vil det utgjøre en *prosess* på din maskin. En slik prosess tildeles en del av internminnet, har sin egen programteller (som holder styr på hvor langt programmet er kommet i eksekveringen) og får tildelt tid i prosessoren ved behov.

Hvis en applikasjon kjører og du prøver å åpne en instans til, skjer forskjellig avhengig av hvordan det er programmert:

- ✓ Noen programmer nekter deg å åpne mer enn én av gangen. De er programmert slik at hvis du prøver å åpne en ny instans, åpner den isteden det vinduet du allerede har åpent. F.eks. er NetBeans programmert slik – det er umulig å åpne NetBeans i to vinduer.
- ✓ Andre programmer tillater flere åpne vinduer og starter en ny prosess for hver gang programmet startes. I "Task Manager" vil du se slike som flere "Applications" og samtidig som flere prosesser. F.eks. er StarUML programmert slik.
- ✓ Noen programmer tillater flere åpne vinduer, men viser allikevel bare én prosess i "Task Manager". Slike programmer kjører flere *tråder* (*threads*) som deler på den ressursen som prosessen har fått. Word og Firefox er f.eks. programmert slik.

En tråd er en egen *eksekveringssti* (*execution path*) innenfor samme prosess. Hver tråd har sin egen programteller og får tildelt tidsressurs på prosessoren, men de deler minnet innenfor det som tildeles prosessen.

En eller flere tråder i et program

De fleste programmer er programmert med én, enkelt hovedtråd for hver prosess. De har da én programteller. Hvis noe tar tid, f.eks. å hente noe over Internett eller fra harddisk, så må programmet vente til oppgaven er utført. Det er ganske vanlig at det blir slik ventetid og det er jo ineffektivt sett fra ditt programs ståsted. Knappen *A* i *DemoTråd* demonstrerer hvordan GUI "henger" når en lengre løkke kjører.

> *Note:* Det er ikke helt sant at det bare er én tråd. De fleste kompilatorer skaper *flere* tråder, for "garbage collection" og annet. Dette er imidlertid tråder vi har liten kontroll over og vanligvis vet lite om. Her snakker jeg om tråder *som vi selv programmerer og starter*. Med knappen *B* i *DemoTråd* kan du se hvilke tråder som kjører når programmet er startet.

Det er da mulig å skrive programmet slik at det *utfører flere algoritmer parallelt*[41]. Det får flere *eksekveringsstier*, dvs. *tråder*. Da kan man legge opp til at det som vil ta tid, utføres i én tråd. Imens fortsetter programmet med andre ting i en annen tråd.

Siden trådene deler minnet (de er i samme prosess), må de koordineres = *synkroniseres*. Hvis ikke kan de lett ødelegge for hverandre, ved at de endrer/bruker samme data.

Klasser i Java er ofte merket *thread-safe* eller *thread-unsafe*. Det dreier seg nettopp om det er mulig for én tråd å ødelegge for en annen. De fleste er nå *thread-unsafe* fordi metodene deres da eksekverer raskere (det er mindre overhead for å sikre mot at tråder ødelegger for hverandre). Noen av dem er også *fail-fast* hvilket innebærer at de umiddelbart oppdager mulige problemer med samtidigheten og da kaster de feil. Typisk er dette når én tråd endrer noe i en samling samtidig som en annen itererer gjennom den. Prinsippene rundt dette er

[41] Uttrykket "parallelt" er litt upresist. Det krever flere prosessorer. Med "multicore" (flerkjerne-)prosessor kan man i prinsippets eksekvere parallelt, men i praksis deler kjernene ressurser som må tidsdeles. Tilsvarende gjelder for maskiner med flere prosessorer ("multi-chip").

svært lik den som gjelder for databaser, der mange klienter parallelt endrer og søker i databasen.

Det er altså to ting å passe på her:
1. Hvordan skaper vi flere tråder i et program, og
2. Hvordan sikrer vi mot feil når trådene aksesserer samme data

Metode 1: Skape flere tråder tradisjonelt med *Runnable* eller *Thread*

Tråder må *instansieres* som objekter. Det vanligste er å implementere grensesnittet *Runnable* og da må metoden *run()* overstyres. Utover dette står man da helt fritt til selv å lage grensesnitt, metoder osv.

En annen mulighet er å arve fra *klassen Thread*. Den implementerer *Runnable* og har ferdige metoder som *start*, *isAlive*, *getState*, *interrupt* osv. Også der må man overstyre *run()*. Grunnen til at dette er mindre vanlig er at man ofte ønsker å arve fra andre klasser og Java tillater som kjent ikke multippel arv (fra flere klasser). I *DemoTråd* demonstrerer knapp *C* hvordan to tråder startes og skriver til *System.out*. GUI reagerer fortsatt, f.eks. virker fortsatt knappen *X* oppe til høyre i GUI.

Eksempel:
Jeg lager først en klasse *Teller* som implementerer *Runnable*. Jeg implementerer metoden *run()* og lager også en *toString()* så jeg lett kan skrive ut til *System.out*. Inne i løkken i *run()* legger jeg inn en pause, slik at andre tråder kan slippe til – hvis jeg ikke gjør det, så vil den ene gjøre seg ferdig før den andre slipper til (det er jo faktisk mest effektivt slik fordi det er en del overhead for operativsystemet med å stoppe én tråd og starte en annen):

```
public class Teller implements Runnable{
    private String navn;
    private int tall = 0;
    public Teller(String navn) {
        this.navn = navn;
    }
    @Override
    public synchronized void run() {
       while (tall<=100){
            System.out.println(toString());
            DemoTrådView.liste.add(tall);
            tall++;
            //Be denne tråden vente litt
            try {
                Thread.sleep(20);
            }
            catch (Exception ex) {
                    //Oops – aldri tom catch, men altså…
            }
        }
    }
    @Override
    public String toString() {
        return "Teller{" + "navn=" + navn + ", tall=" + tall + '}';
    }
}
```

I hovedprogrammet skaper jeg to slike tråder *Teller* og starter dem med *start()* som er startmetoden for *Thread*. Den kaller trådens *run()*. Dette er demonstrert med knappen *C* i *DemoTråd*.

```
private void butStartActionPerformed(java.awt.event.ActionEvent evt) {
    Thread tråd1 = new Thread(new Teller("Teller nr 1"));
    Thread tråd2 = new Thread(new Teller("Teller nr 2"));
    //Starter trådene
    tråd1.start();
    tråd2.start();
}
```

De to trådene alternerer og skriver noe slikt ut på *System.out* (vi har ikke kontroll på rekkefølgen i utskriften da det er operativsystemet som prioriterer den ene eller den andre):

```
Teller{navn=Teller nr 2, tall=0}
Teller{navn=Teller nr 1, tall=0}
Teller{navn=Teller nr 2, tall=1}
… osv.
```

Vi kunne også bedt trådene om å vente i hovedprogrammet, men da blir ikke knappens hendelsesprosedyre avsluttet, og da "henger" GUI så lenge.

Dette ville vært temmelig likt om vi hadde laget klassen *Teller* som en subklasse av *Thread*.

Synkronisering

For å unngå at den ene tråden ødelegger for den andre hvis de aksesserer samme data, kan vi gjøre metoden som aksesserer dataene *synkronisert* (*synchronized*). Vi føyer til ordet foran metoden:

```
public synchronized void run() {
… osv. som før
```

Det skal medføre at bare én tråd får kjøre *run* av gangen. Dermed vil de ikke kollidere når de oppdaterer samme variabel. (Det går ofte bra uten også.)

Oppdatering av GUI med bruk av tråder

Når tråder skaper resultater som man vil vise i GUI, får man problemer. Av de trådene som kjører, er det bare én som får oppdatere GUI, nemlig det vi oppfatter som "hovedprogrammet". Det kalles *Event Dispatch Thread* (EDT). Vi trenger altså en kommunikasjon fra trådene til EDT-tråden. Det er vanligvis vanskelig, og JavaDoc anbefaler at man lar det være.

Det er allikevel mulig, hvis man skaper hele trådklassen som anonymt objekt i hovedtråden. Da er GUI-komponentene tilgjengelige (de deklareres *private* og er utilgjengelige fra en utenforstående trådklasse).

JavaDoc sier det slik[42] (min uthevelse):

> *When writing a multi-threaded application using Swing, there are two constraints to keep in mind: (refer to How to Use Threads for more details):*
>
> - *Time-consuming tasks should not be run on the Event Dispatch Thread. Otherwise the application becomes unresponsive.*
> - *Swing components should be accessed on the Event Dispatch Thread only.*
>
> *These constraints mean that a GUI application with time intensive computing needs at least two threads: 1) a thread to perform the lengthy task and 2) the Event*

[42] http://docs.oracle.com/javase/8/docs/api/javax/swing/SwingWorker.html

> *Dispatch Thread (EDT) for all GUI-related activities. **This involves inter-thread communication which can be tricky to implement.***
>
> *SwingWorker is designed for situations where you need to have a long running task run in a background thread and provide updates to the UI either when done, or while processing. Subclasses of SwingWorker must implement the doInBackground() method to perform the background computation.*

Vi må altså se nærmere på SwingWorker-klassen.

Metode 2: SwingWorker (som ofte er greiere å bruke enn *Runnable* og *Thread*)

Når vi skal starte noe som vil ta tid, og som skal oppdatere GUI – enten mens den pågår eller etterpå når den er ferdig, passer det med *SwingWorker*. *SwingWorker* kan starte flere tråder.

SwingWorker er en abstrakt klasse som vi kan bruke i arv til å lage våre egne klasser. Man kan om ønskelig angi to typer:

```
SwingWorker<T,V>
```

Typen *T* er typen data som metodene *doInBackground* og *get* returnerer. Typen *V* er den typen som metodene *publish* og *process* bruker. Hvis vi ikke angir typer, brukes *<Object, Object>* og det er ofte like greit, men gir mindre kompilatorkontroll.

Vi definerer da en trådklasse som arver *SwingWorker*. Da har vi en rekke krav og muligheter:

1. Vi kan selvsagt deklarere *felt* (variable) og *tilgangsmetoder* i klassen etter behov.
2. Vi kan definere *konstruktører*, gjerne med parametre. Ofte klarer det seg med den parameterfrie default-konstruktøren som kompilatoren automatisk legger til.
3. Vi *må* definere *doInBackground()* som er den metoden som startes fra hovedtråden med metoden *execute()*. Det er her hovedarbeidet skjer – det som forventes å ta lang tid. Det er en funksjon som returnerer en verdi (evt. *null* hvis du ikke har bruk for den i hovedtråden). Returverdien hentes i hovedprogrammet med *get()*, men da må du være klar over at hovedtråden "henger" ved *get()* og venter på returen[43].
4. Vi kan definere andre metoder, avhengig av hva vil oppnå:
    a. *done()*. Den startes i hovedtråden når *doInBackground* er ferdig. De vil allikevel ikke ha tilgang til komponentene som alltid er deklarert *private* i *NetBeans*, unntatt når den deklareres/defineres anonymt i hovedtråden.
    b. *publish()* og *process()*. Metoden *publish()* sender data til metoden *process()* som kjører i EDT og kan oppdatere GUI (hvis den har tilgang). Jeg har ikke brukt disse og kjenner ikke alle detaljene i hvordan de virker.

For å få oppdatert GUI, må *SwingWorker*-tråden kommunisere med hovedtråden EDT. Det gjøres med hendelser av typen *PropertyChange*. Vi endrer altså en egenskap ved tråd-objektet, og det fyrer en *PropertyChange*-hendelse som en lytter i hovedtråden kan fange opp. Lytteren knyttes til tråd-objektet som ble skapt i hovedtråden (og startet med *execute*).

Det er to forhåndsdefinerte egenskaper som vil sende en hendelse til lytteren hvis de endres:

1. *state* som angir trådens tilstand. Den har verdier som følger:
    a. *SwingWorker.StateValue.PENDING* – tråden er i ferd med å starte, men jobber ennå ikke
    b. *SwingWorker.StateValue.STARTED* – tråden er i gang med *doInBackground*

---

[43] GUI "henger" også – du må vurdere om det er slik du ønsker brukergrensesnittet. Det kan det jo være, hvis du vil hindre brukeren i gjøre noe mens tråden kjører, samtidig som du vil kjøre *mer enn én tråd* ved siden av hovedtråden. Hvis det ikke er aktuelt med flere sidetråder og brukergrensesnittet skal henge, kan du like gjerne ha den tunge algoritmen i hovedtråden.

c. *SwingWorker.StateValue.DONE* – tråden har eksekvert *doInBackground* ferdig og her også ferdig med metoden *done*

i denne rekkefølgen. Det er tråden selv som endrer denne egenskapen, du kan ikke programmere det selv.

2. *progress* som angir hvor langt tråden er kommet med arbeidet. Denne verdien må du selv sette i programmet, med tilgangsmetoden *setProgress(int progress)* der *progress* er et heltall i intervallet [0..100].
3. En hvilken som helst egenskap som du selv finner nyttig. Du lager ikke egentlig egenskapen, men sender hendelsen med et navn med

```
void firePropertyChange
     (String propertyName, Object oldValue, Object newValue)
```

De to verdiene *oldValue* og *newValue* må være forskjellige ellers har det jo faktisk ikke skjedd en endring. Hvis du bare skal bruke den ene, kan du bare sette den andre til noe som aldri forekommer, f.eks. *null* e.l.

I hovedprogrammet skaper du en *PropertyChangeListener* som lytter etter endringer i trådens egenskaper. Avhengig av hvilken endring det er, oppdaterer du GUI eller foretar deg andre handlinger i hovedtråden. Dette vil bare gi ønsket effekt hvis hovedtråden er ledig – f.eks. at den ikke venter med *sleep* eller *get*. Hendelsene som tråden rapporterer, blir satt i hendelseskøen for at hovedtråden EDT skal ta seg av dem ved første leilighet. Trådene kan gis prioritet, men det kan fort gi uønskede resultater.

Nedenfor – hentet fra programmet *DemoTråd* ser du hvordan jeg først har definert en *SwingWorker*, instansiert den som et tråd-objekt og startet den. Jeg lager også et lytterobjekt som knyttes til tråd-objektet og gjør forskjellige ting avhengig av hvilken hendelse som kommer. Lytteren er skapt anonymt og blir en del av hovedprogrammet (det kan også gjøres som en indre klasse med samme effekt) og har da tilgang til hovedprogrammets private variable og private komponenter.

Et litt større eksempel

Tråden

Jeg har her fulgt forklaringene ovenfor (hendelsesmetode for knappen D i DemoTråd) og laget nedenstående tråd-klasse. Den skal telle til 1000 og underveis rapportere hvor langt den er kommet til EDT (*setProgress*) og sende tallet den holder på med i øyeblikket (*firePropertyChange*). Hovedtråden skal oppdatere GUI med disse verdiene. Jeg har da laget min egen egenskap "iterator" – navnet har jeg selv funnet på.

Underveis skriver også tråden seg selv til *System.out*. Jeg "gadd" ikke lage *toString* her.

Videre skal tråden vise en meldingsboks når den er ferdig. Det er programmert i metoden *done*.

```
public class TellerTråd
        extends SwingWorker {
    String navn; //for å identifisere denne tråden

    public TellerTråd(String navn) {
        this.navn = navn;
    }
```

```
    @Override
    public Object doInBackground() {
        //Dette kjøres når tråden startes med execute()
        //Det fortsetter å kjøre til det er ferdig
        int framdrift;
        final int MAKS = 1000;
        for (int tall=0;tall<=MAKS;tall++){
            System.out.println(navn + ": " + Integer.toString(tall));
            //rapporeter fremdrift til EDT
            framdrift = tall*100/MAKS; //OBS! regnes i prosent [0..100]
            setProgress(framdrift);
            //send en hendelse kalt "iterator"+navnet til EDT
            firePropertyChange("iterator", -1, tall);
            //Be denne tråden vente litt
            try {
                Thread.sleep(10);
            }
            catch (Exception ex) {
            }
        }
        //Returen plukkes opp av get() men da stanser hovedprgrammet og
        //GUI henger
        return "Retur fra tråd: " + navn + " er ferdig ";
    }

    @Override
    protected void done(){
        //Dette kjøres i EDT etter at tråden er ferdig med
doInBackground
        //GUI i EDT er ikke synlig her fordi de er private
        JOptionPane.showMessageDialog(null, "Rapport fra done(): "
                + navn + " er ferdig");
    }
}
```

Lytteren

Jeg trenger en lytter som knyttes til de to trådene i hovedprogrammet. Denne lytteren har jeg definert som en *indre klasse*[44], og den ser slik ut:

```
    private class TrådLytter implements PropertyChangeListener{
        //En indre klasse for en lytter som lytter på tråder av typen
TellerTråd

        TellerTråd minTråd; //for å kunne bruke get() på riktig tråd
        JLabel lblTeller;  //for å vite hvilken etikett som skal
oppdateres

        public TrådLytter(TellerTråd minTråd, JLabel lblTeller){
            this.minTråd=minTråd;
            this.lblTeller=lblTeller;
        }
```

[44] I kapittel 1, da vi laget grensesnittet manuelt, ble det anbefalt å lage lyttere som indre klasser. Det er mulig å bruke eksterne klasser, men det er betydelig vanskeligere. Vanskeligheten oppstår fordi lytteren da ikke har tilgang til hovedklassens komponenter.

```
        @Override
        public void propertyChange(PropertyChangeEvent evt) {
            if ("progress".equals(evt.getPropertyName())) {
                lblProgress.setText(evt.getNewValue().toString());
            }
            if ("iterator".equals(evt.getPropertyName())){
                lblTeller.setText(evt.getNewValue().toString());
            }
            if (evt.getNewValue()==
                        SwingWorker.StateValue.DONE){
                //Tråden er ferdig (og har også utført done)
                try {
                    lblFerdig.setText
                        (lblFerdig.getText()
                        + (String)minTråd.get());
                }
                catch (Exception ex) {
                }
            }
        }
    }
```

I metoden *propertyChange* som må overstyres, trenger jeg å oppdatere riktig *JLabel*. Jeg bruker et parameter i konstruktøren til å få beskjed om hvilken det er. Jeg kan ikke se hvordan jeg enkelt skulle unngå det.

Videre skal jeg bruke *get()* på riktig tråd, og da må jeg også vite hvilken tråd denne lytteren er tilknyttet. Det setter jeg også i konstruktøren. Isteden kunne jeg nesten nederst ha erstattet

```
(String)minTråd.get()
```

med det mer komplekse, men også mer generelle

```
(String)((TellerTråd)evt.getSource()).get()
```

Der angir *evt.getSource* hvilket objekt som skapte hendelsen. Det kildeobjektet gjør jeg om til en *TellerTråd* og da kan jeg bruke *get()* på den. Det hele omgjøres så til en streng for å gi riktig type til *setText*.

Legg merke til at jeg sjekker hvilken hendelse som ankommer ved å sammenlikne strengen som ble sendt med hendelsene og gjør forskjellige ting avhengig av hvilken hendelse det er.

Hovedprogrammet (EDT)
Hovedprogrammets hendelsesmetode ser ut som nedenfor:

```
    private void butTellActionPerformed(java.awt.event.ActionEvent evt)
{
        //Viser bruk av to tråder som kommuniserer til EDT i
hovedprogrammet
        //med hendelser

        //Skap tellertråd nr 1 med lytter
        TellerTråd tråd1 = new TellerTråd("Tråd 1");
        tråd1.addPropertyChangeListener(new
TrådLytter(tråd1,lblTeller1));

        //Skap en tellertråd til på samme måte
        TellerTråd tråd2 = new TellerTråd("Tråd 2");
        tråd2.addPropertyChangeListener(new
TrådLytter(tråd2,lblTeller2));

        //Start trådene og avslutt hendelsesmetoden
        //så hovedprogrammet kommer videre.
        tråd1.execute();
        tråd2.execute();
    }
```

Oppgave til kapittel 12

Denne oppgaven er hentet fra en "Thursday Code Puzzler" fra DZone (http://java.dzone.com/articles/thursday-code-puzzler-sieve). Det er mye moro på DZone i tillegg til de fine RefCardz (http://refcardz.dzone.com/).

Et primtall er et naturlig tall (1, 2, 3 osv.) større enn 1 som bare kan deles med seg selv og med 1 uten å gi rest. Vi skal finne alle primtall som finnes blant de første *n* tallene (*n* oppgis av brukeren).

> *Note:*
> Primtall er viktige for hashing, fordi hashingalgoritmer som involverer primtall sprer hashtallene best, derved fylles "bøttene" jevnt opp. I NetBeans brukes alltid minst ett primtall som en del av algoritmen nåe NetBeans lager *hashCode*.
>
> Videre er primtall viktig for kryptering. Krypteringssystemet RSA[45] også kalt "public key, private key" er meget brukt. Det tar utgangspunkt i to primtall med minst 100 siffer hver. Disse to primtallene ganges med hverandre og danner den ene nøkkelen.

Det finnes flere algoritmer for å finne primtall. En mulighet er å forsøke å dele tallet (her kalt *x*) med 2, 3, 4 osv. opp til *x-1*. Hvis det ikke finnes noen blant dem som gir rest lik 0, så er tallet et primtall. F.eks. kan vi prøve å dele 11 med 2, 3, 4...10 og finner at ingen av dem "går opp". Altså er 11 et primtall. Dette er jo en grei algoritme å programmere, men vil ta lenger å lenger tid etter hvert som tallet *x* stiger. Det vil f.eks. ta ganske lang tid å sjekke om 10.000.001 er primtall på denne måten. En optimalisering vil være å hoppe ut med en gang vi finner et tall som "går opp" i tallet *x*. Vi kan også stoppe ved *x/2*, men fortsatt blir det tungt.

---

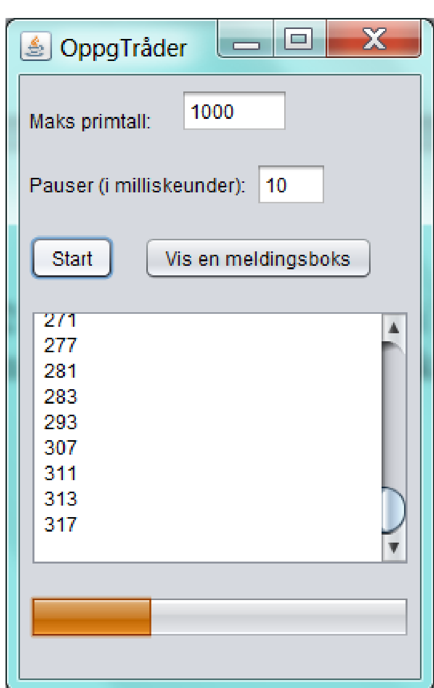
[45] RSA ble først beskrevet offentlig i 1977 av **R**ivest, **S**hamir og **A**dleman med MIT – derav navnet. Matematikeren Cock hadde tidligere beskrevet noe lignende i hemmelighet for britisk etterretning, offentliggjort i 1997. Krypteringsprogrammet PGP er et liknende system. På grunn av USAs spesielle eksportlover ble programvaren eksportert ved at den ble trykket i en bok, satt på et fly og så scannet inn igjen i Europa. USA har jo selvsagt trykkefrihet!

En bedre – og i alle fall morsommere og flere tusen år gammel metode – kalles "Eratosthenes sil". En gøyal, dynamisk forklaring[46] finner du på http://www.youtube.com/watch?v=V08g_lkKj6Q. Du finner også en god forklaring på norsk på WikiPedia (https://no.wikipedia.org/wiki/Eratosthenes%27_sil).

Fordi dette kan være en langvarig jobb, skaper vi en tråd som kjører algoritmen. Det vil være fint å få primtallene skrevet inn i en *TextArea* etter hvert som de finnes. Hvis vi regner med at vi skal sjekke ett og ett tall fra 2 til *n* kan vi også oppdatere en *JProgressBar* fra 0 til 100 mens tråden går. Metoden *setProgress(framdrift)* kan jo bare ha argument opp til 100, så hvis variabelen *x* er det tallet vi sjekker akkurat nå, så regnes argumentet *framdrift* som *x* i prosent av *n*.

La brukeren fylle inn *n* og evt. pause for hver iterasjon og bruk disse verdiene som argumenter når tråden skapes.

Lag en knapp som starter beregningene og ha annen knapp som bare viser en meldingsboks, så du kan sjekke at GUI virker mens tråden jobber. Hvis tråden går for raskt, kan du legge inn en passende forsinkelse med *sleep*.



Jeg anbefaler å lage en array med objekter for alle tallene fra 2 opp til *n* og ha objekter i listen med en Boolske variabel som angir om tallet er sjekket.

Denne oppgaven er faktisk ganske artig!

Utfordringer:
Prøv å klikke en gang til på knappen "Start" mens den jobber. Hva skjer? Hva kan du gjøre for å unngå det? (Løst i løsningsforslaget.)

Hvis du får tid, så lag en *JSlider* til å angi maks (oppdater en label med valgt verdi) og en *JSpinner* til pause. Da får du trening i å lage lyttere og unngår at knappen "Start" kaster feil.

[46] Merk at *composite number* (sammensatt tall) er et tall som *ikke* er primtall. En artig regel er at absolutt alle sammensatte tall er et produkt av to eller flere primtall.

Kapittel 13 – Noen nyttige språkelementer

I dette kapittelet tar jeg for meg noen nyttige strukturer/språkelementer som er kommet til i senere Java-versjoner. Generelt utvides Java stadig. Jeg tror det har sammenheng med at Sun først var opptatt av at kompilatoren skulle være enkel og ha liten "footprint" (størrelse av den eksekverbare filen) og at bytekoden skulle være tilsvarende liten. Det var nyttig da Java primært ble benyttet over nett, særlig i form av Applets. Etter hvert brukes Java mer og mer til vanlig applikasjonsprogrammering, og da betyr størrelse/enkelhet mindre, og det blir viktigere med fleksibilitet og muligheter for programmereren. Mange av disse "nyhetene" bør en Java-programmerer kjenne til. Her omtaler jeg et lite utvalg.

Streams

I Java 8 er det kommet et nytt bibliotek kalt *java.util.stream*. Den inneholder flere klasser og en av dem er klassen *Streams*. I tillegg er det et antall metoder for å manipulere med strømmen. Videre har en del samlinger fått metoder som skaper strømmer – Collection omgjøres f.eks. med metoden *stream()* til en strøm.

En strøm kan sees på som verdier (objekter) kjedet sammen sekvensielt. Her er noen egenskaper ved strømmer:

- ✓ I motsetning til en Collection, der elementene har verdi, så skapes en strøm "on the fly". Verdiene i strømmen lagres ikke.
- ✓ Metodene som brukes, endrer ikke verdien i strømmen på noen måte, men skaper isteden en ny strøm. Hvis f.eks. en strøm filtreres, så endres ikke den opprinnelige strømmen, men en ny (mindre) strøm skapes.
- ✓ Strømmer kan være ubegrenset i størrelse.
- ✓ Operasjonene deles i
  - o "mellomliggende" (*intermediate*) som resulterer i en ny strøm, f.eks. *sorted* og *filter*
  - o "avsluttende", "terminale" (*terminal*) som produserer en verdi (ikke en strøm) f.eks. *forEach* og *findFirst.*
- ✓ Operasjonene kan være "lazy", dvs. at de hopper ut så snart et resultat er funnet. Dette er også helt nødvendig siden strømmen kan være ubegrenset stor.
- ✓ Siden operasjonene resulterer i strømmer, kan de kjedes slik at output fra den ene brukes som input til den neste. Det kalles pipelining og er et gammelt prinsipp i andre sammenhenger[47]. I en pipeline starter man med en strøm av verdier fra en kilde som input til en operasjon, output fra denne operasjonen sendes videre som input til neste operasjon osv. som på et samlebånd. Prosessen avsluttes med en terminal operasjon som faktisk produserer sluttresultatet. Sluttresultatet er ikke en strøm men en enkeltverdi eller samling og kan derfor ikke pipes videre. "Røret" som verdiene sendes gjennom er anonymt og kan ikke navngis.
- ✓ Noen av operasjonene er
  - o "stateless" dvs. de tar ikke vare på verdier som de allerede har sett. Operasjonen *filter* er slik.
  - o "stateful" dvs. de er avhengig av å ta vare på verdier som er ferdigbehandlet. Operasjonen *distinct* er slik.
- ✓ Noen operasjoner er dessuten "kortsluttet" (short circuited). Det defineres som at den produserer et endelig resultat (en strøm eller en sluttverdi) også når strømmen er uendelig.

[47] Første gang jeg selv traff på det var i Unix og det var faktisk der det ble oppfunnet. MS DOS tok snart opp idéen implementert som "uekte" pseudo-strømmer og siden har mange brukt den – også databaser. Du kan lese mer om prinsippet på https://en.wikipedia.org/wiki/Pipelining.

Man kan produsere strømmer f.eks, slik

```
Collection<Integer> samling = new ArrayList<>();
samling.add(5);
samling.add(4);
Stream minStrøm = samling.stream(); //returnerer en Stream
minStrøm = minStrøm.sorted();
minStrøm.forEach(x -> System.out.println(x));
```

Dette vil skrive tallene 4 og 5 (sortert) til output. Den siste linjen er eksempel på Lambda-uttrykk – se nedenfor.

Istedenfor å bruke variabelen *minStrøm* ovenfor, kan man skrive direkte:

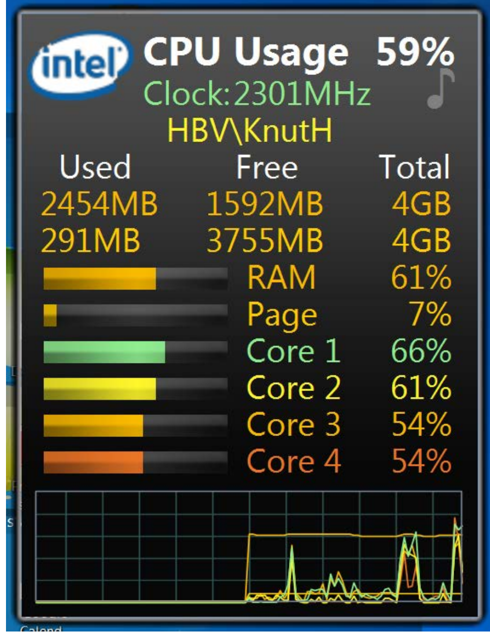
```
Collection<Integer> samling = new ArrayList<>();
samling.add(5);
samling.add(4);
samling.stream().sorted().forEach(x -> System.out.println(x));
```

Denne gjør det samme som algoritmen ovenfor men uttrykt mer kompakt. Uttrykket utnytter at en strøm som behandles kan resultere i en ny strøm som kan viderebehandles.

Java 8 har fått forbedrede mekanismer for parallellprosessering (jobben deles i flere biter som startes på hver sin prosessor). Strømmer støtter dette ved at man skaper flere strømmer f.eks. slik:

```
System.out.println(samling.parallelStream().count());
```

Denne kan kjøre flere strømmer parallelt. Strømmen deler seg (*fork*) og hver delstrøm teller "sine" verdier hver for seg. Når alle er ferdige med sin del av jobben, samles de (*join*) og produserer ett, samlet resultat. Her kan du se at alle mine fire prosessorkjerner (*Core 1* osv.) er aktive med på dette (jeg har lagt til mange elementer i *samling* så det tar litt tid):



JavaDoc for Stream: http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Lambda expressions

Lambda er lagt til i Java 8 for (a) å støtte intern iterasjon nå og (b) senere for å støtte funksjonell programmering. Lambda gjør det også enklere å prosessere tunge algoritmer parallelt.

Lambda uttrykk er utrykk som gir en verdi, gjør noe (har sideeffekt) eller begge deler. Et Lambda-utrykk kan se slik ut:

```
(a, b) -> a + b;
(a, b) -> {return a + b;}
x -> System.out.println(x)
```

Det bygges opp slik:

a. En liste med variable, adskilt med komma og omsluttet av parenteser.
   Hvis det bare er én variabel er det ikke nødvendig med parenteser.
   Det er ikke nødvendig å angi variablenes type.
b. Arrow token som skrives `->`.
   Jeg like å lese det som "send til" men det betyr egentlig at variablene listet opp foran (pkt a) skal anvendes i uttrykket som følger etter (pkt c).
c. Et uttrykk der variablene anvendes.
   Uttrykket evalueres og returneres.
   Uttrykket kan også inneholde setninger, men da som en blokk omsluttet av spissklammene { og }. Hvis det bare er én setning som returnerer void, kan man utelate spissklammene.

Lambda kan brukes i strømmer – et eksempel ble vist for strømmer ovenfor. Her er et annet eksempel:

```
samling.stream().filter(x -> x % 2 == 0).forEach(x ->
System.out.println(x));
```

I dette eksemplet dannes det en strøm av *samling*. Den filtreres slik at bare de verdiene i strømmen som tilfredsstiller `x % 2 == 0` dvs de er partall, godtas. Metoden *filter* forventer en *boolean* og tar bare med de verdiene i strømmen som returnerer *true*. Deretter sendes strømmen videre til *forEach* som skriver ut dem som kommer igjennom filteret. Resultatet er at bare partall i *samling* skrives ut.

Videre kan Lambda benyttes sammen med grensesnitt (interface). Her er et eksempel på det. Jeg starter da med å deklarere grensesnittet:

```
interface Kalkulator {
    double operasjon(double a, double b); //Kun deklarasjon
}
```

Grensesnittet *Kalkulator* er laget som et *funksjonelt grensesnitt*, dvs. et grensesnitt med bare én abstrakt metode[48]. Når Lambda skal brukes slik vist som nedenfor, kreves det bruk av funksjonelle grensesnitt.

Deretter instansierer jeg grensesnittet og definerer grensesnittets funksjon *operasjon* med Lambda-uttrykk:

```
Kalkulator addisjon = (a, b) -> a + b;
Kalkulator subtraksjon = (a, b) -> a - b;
Kalkulator multiplikasjon = (a, b) -> a * b;
Kalkulator divisjon = (a, b) -> a / b;
```

Lambda-utrykket oppfattes som definisjonen av grensesnittets funksjon *operasjon*. Siden grensesnittet bare har én funksjon, kan jeg la være å oppgi navnet på den (kompilatoren vet jo hvilken funksjon det dreier seg om).

Nå kan jeg bruke disse variablene, som har *hver sin* definisjon av funksjonen *operasjon*:

```
System.out.println(addisjon.operasjon(7, 2));
System.out.println(subtraksjon.operasjon(7, 2));
System.out.println(multiplikasjon.operasjon(7, 2));
System.out.println(divisjon.operasjon(7, 2));
```

[48] Se allikevel nedenfor om statiske (klasse-)metoder og default-metoder.

Systemer skriver ut svarene på beregningene:
```
run:
9.0
5.0
14.0
3.5
```

Det hadde ikke vært noe i veien for å bruke samme variabelen til alt:
```
Kalkulator beregn = (a, b) -> a + b;
System.out.println(beregn.operasjon(7, 2)); //skriver 9.0
beregn = (a, b) -> a - b;
System.out.println(beregn.operasjon(7, 2)); //skriver 5.0
//...osv
```
Jeg redefinerer her stadig funksjonen *beregn.operasjon*.

Istedenfor Lambda, kunne jeg brukt grensesnittet Kalkulator som tilordning med *new*[49], slik:
```
Kalkulator potens = new Kalkulator() {
    @Override
    public double operasjon(double a, double b) {
        return Math.pow(a, b);
    }
};
System.out.println(potens.operasjon(7, 2)); //Skriver 49.0
```
Sammenliknet med setningen `Kalkulator addisjon = (a, b) -> a + b;` ovenfor, ser du at Lambda-uttrykket har erstattet *new Kalkulator* med definisjon av den abstrakte metoden *operasjon*.

Det finnes ingen direkte JavaDoc for Lambda som jeg kan finne, men en meget grundig innføring finnes på http://www.lambdafaq.org/.

Klasse-metoder (static) i grensesnitt

I Java 8 kom muligheten til å definere klasse-metoder i grensesnitt. De kan ikke overstyres av implementerende klasser. Her er et svært enkelt eksempel:
```
public interface Grensesnitt {
    public static boolean erSann(){
        return true;
    }
}
public class Impl implements Grensesnitt {
    public static boolean erSann() {
        return false;
    }
}
```

Metoden *erSann* i klassen *Impl* overstyrer *ikke* metoden *erSann* fra grensesnittet, da det fortsatt er mulig å skrive både
```
Grensesnitt.erSann();
```
og
```
Impl.erSann();
```
De vil gi forskjellig resultat.

Det samme gjelder for øvrig ved arv av klasse-metoder.

Generelt gjelder overstyring bare for objekt-medlemmer.

[49] Det er tillatt å bruke *new* på et grensesnitt hvis man *samtidig* definerer *alle* abstrakte metoder. Hvis man bare prøver å instansiere et grensesnitt med *new* som f.eks. Kalulator ulovlig = new Kalkulator(); så vil kompilatoren gi feilen *Kalkulator is abstract - cannot be instantiated*

Default-metoder i grensesnitt

En annen interessant utvidelse i JDK 8 er at grensesnitt kan ha *default-metoder* også kalt *defender methods* eller *virtual extended methods*. Det er metoder som er definert[50] i grensesnittet (det var ikke lov tidligere) og som definerer metoden i mangel av overstyring. Tidligere kunne ikke grensesnitt *definere* – bare *deklarere* – metoder og de ble dermed alltid abstrakte[51].

I eksemplet *interface Kalkulator* ovenfor, kan jeg legge til en default-metode uten at grensesnittet av den grunn mister sin egenskap som funksjonelt grensesnitt:

```
interface Kalkulator {
    double operasjon(double a, double b);
    default double størst (double x, double y){
        return (x > y? x: y); //returner største tallet
    }
}
```

Når jeg da senere skriver

```
Kalkulator addisjon = (a, b) -> a + b;
```

som før, så er det fortsatt opplagt at det er *Kalkulator.operasjon* jeg definerer. På den annen side kan jeg nå også skrive

```
System.out.println(beregn.størst(7,2)); //skriver 7.0 med default-
metoden
```

Et *grensesnitt* som *arver* fra *Kalkulator* (*extends*) kan velge mellom følgende alternativer for default-metoden:

1. Ikke si noe om default-metoden. Da arver den default-metoden slik den er definert i *Kalkulator*.
2. *Redeklarere* default-metoden (uten definisjon). Da blir den abstrakt.
3. *Redefinere* default-metoden (med ny algoritme). Da blir den overstyrt.

Samtidig kan vi velge å definere den abstrakte metoden *beregn*, men da må den merkes *default*. Altså:

I *grensesnitt* må alle objekt-metoder enten være udefinerte (abstrakte) eller merkes *default*. Definerte klasse-metoder er også tillatt og de kan ikke overstyres.

Videre kan grensesnitt fortsatt definere konstanter (*final*) – både klasse-konstanter (*static*) og objekt-konstanter – men ikke variable.

En *klasse* som *implementerer* dette grensesnittet (*implements*), må definere metoden *beregn* eller gjøres abstrakt. Default-metoden kan overstyres om ønskelig, ellers blir den som angitt i *Kalkulator.* Dette er akkurat som vanlig.

Dette opplever jeg som en fin og nyttig nyhet. Nå savner jeg egentlig bare *optional parameters* så er Java kommet langt!

Les om default-metoder på
http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html.

---

[50] Jeg minner igjen om forskjellen på en *deklarasjon* der bare metoden signatur angis, uten "kropp" altså algoritme og – på den annen side – *definisjon* der "kroppen" er med og inneholder algoritmen som definerer metodens virkemåte.

[51] I *grensesnitt* var alle metoder abstrakte og det var derfor ikke vanlig (men lovlig) å merke dem *abstract*. I *klasser* må abstrakte metoder merkes med *abstract*.

Fork/join

Dette kom med Java 7. Det skal gjøre parallellprosessering enklere ved at man dele et arbeid på flere prosessorer uten å skape tråder eksplisitt. Ovenfor (side 117) viste jeg hvordan strømmer kan støtte slik optimalisering. Her skal jeg vise det mer eksplisitt med *fork/join*.

Algoritmen er *rekursiv* og ideen er som følger:
compute() =
Regel 1: Hvis arbeidet er enkelt nok, så
utfør arbeidet iterativt
Regel 2: Hvis arbeidet fortsatt er omfattende, så
del arbeidet i to, like store deler og gjennomfør *compute* for hver av dem

Du vil gjenkjenne regel 1 som en "enkel regel" og regel to som en "rekursiv regel" i rekursive algoritmer.

En klasse som gjennomfører algoritmen

Du lager en klasse, f.eks. *MaximumFinder* som skal finne største verdi i en array. Den må implementere *RecursiveTask*, f.eks.:

```
public class MaximumFinder extends RecursiveTask<Integer>
```

der *RecursiveTask* er en typet, abstrakt klasse som kjører en algoritme (gitt i metoden *compute*) rekursivt og gir et resultat.

Klassen har de feltene som er nødvendige, her:

```
private final[52] int[] data; //arrayen som skal undersøkes
private final int start; //hvor i arrayen letingen skal starte
private final int end; //..og hvor den skal slutte
```

Du lager to konstruktører, én for det første kallet fra "utsiden" og én for rekursive kall "innefra". Begge setter bare verdier til feltene:

```
private MaximumFinder(int[] data, int start, int end) { //fra innsiden
    this.data = data;
    this.start = start;
    this.end = end;
}
public MaximumFinder(int[] data) { //fra utsiden
    this(data, 0, data.length);
}
```

Den siste av disse vil lete fra element 0 til slutten, den første leter i en del av arrayen.

Du må definere metoden *compute* som angitt i den rekursive algoritmen ovenfor, f.eks. slik

```
    @Override
    protected Integer compute() {
        // 1: Enkel regel: Avslutt rekursjonen
        // OBS! Normalt vil man ikke splitte helt ned til dette nivået.
        // Det er mer effektivt å stoppe når det bare er noen igjen,
        // f.eks. 100 til 10.000 stk og da finne svaret iterativt
        int length = end - start;
        if (length == 1)
            return data[start];
```

[52] *final* gir noe optimalisering, da kompilatoren og JVM vet at referansen til objektet (eller verdien av primitiver) ikke kan endres. Da kan verdien f.eks. lagres i en cache i prosessoren. Det gir rask tilgang og her er vi jo ute etter å optimalisere eksekveringen på fart.

```
        // 2: Rekursiv regel: Del jobben i to rekursivt
        final int split = length / 2;
        final MaximumFinder left = new MaximumFinder
           (data, start, start + split);
        // 2a: Start left asynkront
        left.fork();
        // 2b: Start right asynkront
        final MaximumFinder right = new MaximumFinder(data, start +
split, end);
        // 2c: Vent til begge er ferdige og returner den største
verdien:
        return Math.max(right.compute(), left.join());
    }
```

Legg merke til at man enten (1) avslutter iterativt, eller (2) gjennomfører rekursjon i to deler. I siste tilfelle kalles de to halvdelene av arrayen *left* og *right* og hver av dem skaper et nytt *MaximumFinder*-objekt (det er rekursjonen). *fork* kan bare kalles "innenfra" og starter *compute* asynkront. *join* kan også bare startes "innenfra" og starter også *compute* asynkront. *join* krever at b*egge må være ferdig før programmet kan gå videre*.

Fra hovedtråden

Du begynner med å lage en *ForkJoinPool* og setter den opp med et planlagt antall parallelle løp, f.eks. antall prosessorer på den kjørende maskinen:

```
final int antCores = Runtime.getRuntime().availableProcessors();
final ForkJoinPool pool = new ForkJoinPool(antCores);
```

Det kan godt være aktuelt med flere parallelle enn antall prosessorer, f.eks. hvis algoritmen innebærer venting på eksterne enheter. Maksimalt antall er 32.767. Hvis det ikke er noen venting, vil flere løp bare føre til at de må dele prosessorene og det vil sannsynligvis faktisk gå tregere.

En interessant egenskap med en *ForkJoinPool* er at den tillater "work-stealing", dvs. at én *task* i poolen som er ferdig, kan overta arbeid fra en annen *task* som ikke er det. Derved balanseres arbeidet best mulig. Dette er automatisk og krever ikke noe ekstra kode fra deg.

Deretter skaper vi et *MaximumFinder*-objekt (med den enkle konstruktøren) og starter arbeidet:

```
final MaximumFinder finder = new MaximumFinder(data);
System.out.println(pool.invoke(finder));
```

Argumentet *data* er den arrayen det skal letes i, og *invoke* er det som starter *compute* fra utsiden.

Resultatet

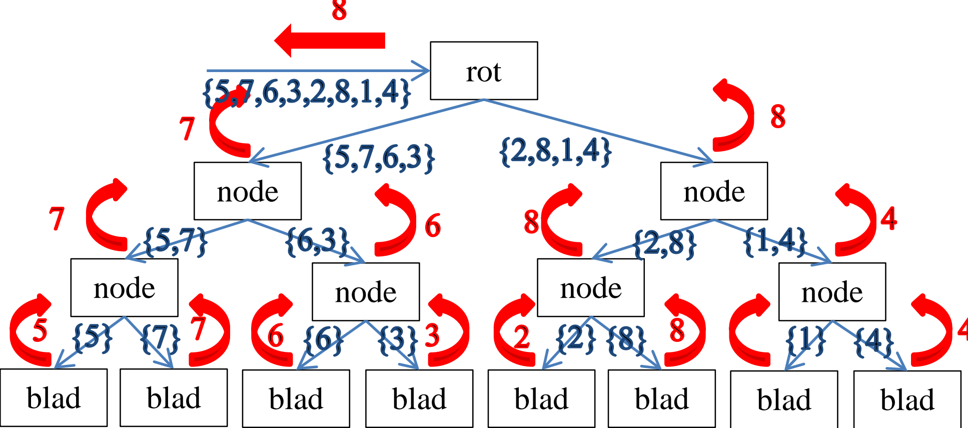
Arbeidet fordeles i en binær trestruktur. Med det som er forklart ovenfor startes et rotobjekt fra "utsiden". Rotobjektet fordeler arbeid på to "barn" som igjen fordeler arbeidet til to "barn" osv. nedover inntil den enkle regelen slår til. Deretter samles resultatet igjen oppover i treet inntil rotobjektet returnerer én verdi (her den største i arrayen).

Diskusjon

Med *fork/join* fikk man en mulighet til å unngå å skape egne tråder. Begrensningen er at arbeidet skaper mange objekter og fyller opp i stakken. Det medfører etter hvert også et stort antall objekter som må skapes og deretter ryddes av garbage collector. Det forenkler arbeidet for programmereren fordi man slipper all kommunikasjon mellom trådene (det er automatisert og skjult), men det har altså en kostnad.

Med *strømmer* som kom i Java 8, kan man gjøre tilsvarende enda enklere med *parallelStream*, men det krever jo at strømmer er tilgjengelig for det objektet man vil

bearbeide i parallelle prosesser. Metoden *parallelStream* bruker *fork/join*-prinsippet rekursivt (som forklart ovenfor), men er jo betydelig enklere å programmere.



Her begynner det med at rotnoden skapes og får arrayen {5,7,6,3,2,8,1,4} (med blått) tilsendt som argument. Rotnoden oppretter to subnoder og sender halvparten av arrayen til hver med *fork*. Disse igjen oppretter to subnoder osv. inntil bladene får en array med bare ett element. Da er det enkelt for dem å finne den største – den de fikk tilsendt må jo være størst – og den returneres til noden over (med rødt). Denne noden venter til den har fått svar fra begge subnodene med *join* og kan da enkelt returnere den største av de to oppover i hierarkiet osv. Tilslutt får rotnoden returnert to verdier og returnerer den største av de to til det kallende objektet.

På hvert nivå må den som er ferdig først, vente til den andre også er ferdig før resultatet som skal returneres, kan bestemmes. Alle bladene kjører den enkle regelen – alle de andre kjører rekursjonsregelen.

Litt om metoder og try-catch

Note: Try-catch og unntakshåndtering har dere hatt før. Her skal jeg gå litt dypere inn i det og diskutere alternativer.

Det kan være fristende å deklarere en metode void. Det kan passe hvis metoden kaster feil når noe går galt. Her gjør den ikke det:

```
public void lesFil (String filnavn) {
     try {
          BufferedReader innLeser
                   = new BufferedReader(new FileReader(filnavn));
          String linje = innLeser.readLine();
          while (linje!=null){
               //TODO:
               //splitt linjen og skap objekt som legges i en samling
               linje=innLeser.readLine();
          }
          innLeser.close();
     }
     catch (Exception ex) {
         //fanger feilen men gjør ikke noe med den
         //Kan ikke vise meldingsboks her
     }
}
```

Det er tre problemer med en slik løsning:

1. Den som kaller metoden får ingen tilbakemelding. Hvis noe skjer og ingen objekter blir skapt, så får ikke den kallende noen beskjed om det.

2. Hvis noe skjer, så kan kallende modul ikke fortelle brukeren hva som har skjedd (f.eks. feil filnavn – prøv igjen).
3. Hvis metoden feiler før *close()* så blir ikke filen lukket. Riktignok "dør" *innLeser* når try er ferdigkjørt fordi den er deklarert lokalt i *try*, og derved fristilles RAM og sikkert filbuffere o.l. også. Men det kan tenkes at filsystemet fortsatt har filen åpen så andre ikke kan få åpnet den.

Man kan prøve å flytte *close()* til *finally*. Da må også *innLeser* deklareres utenfor *try*.

```
public void lesFil (String filnavn) {
    BufferedReader innLeser;
    try {
         innLeser
                 = new BufferedReader(new FileReader(filnavn));
         String linje = innLeser.readLine();
         while (linje!=null){
             //TODO:
             //splitt linjen og skap objekt som legges i en samling
             linje=innLeser.readLine();
         }
    }
    catch (Exception ex) {
        //fanger feilen men gjør ikke noe med den
        //Kan ikke vise meldingsboks her
    }
    finally {
        innLeser.close();
    }
}
```

Problemet da er at

1. *close* kan også kaste feil, f.eks. hvis filen ikke er åpnet. Hvis poenget var å unngå feil, må også den i *try-catch*. Det hjelper ikke å prøve med *innLeser.ready* for den gir false når filen er ferdig lest.

```
public void lesFil (String filnavn) {
    BufferedReader innLeser;
    try {
         innLeser
                 = new BufferedReader(new FileReader(filnavn));
         String linje = innLeser.readLine();
         while (linje!=null){
             //TODO:
             //splitt linjen og skap objekt som legges i en samling
             linje=innLeser.readLine();
         }
    }
    catch (Exception ex) {
        //fanger feilen men gjør ikke noe med den
        //Kan ikke vise meldingsboks her
    }
    finally {
        try {
            innLeser.close();
        }
        catch (Exception ex) {
            //Gjør ikke noe her heller
        }
    }
}
```

Nå begynner det å bli litt rotete og to problemer gjenstår:

1. Den som kaller metoden får ingen tilbakemelding. Hvis noe skjer og ingen objekter blir skapt, så får ikke den kallende noen beskjed om det.
2. Hvis noe skjer, så kan kallende modul ikke fortelle brukeren hva som har skjedd (f.eks. feil filnavn – prøv igjen).

En bedre løsning

Problemene ovenfor oppstår fordi man ikke er villig til å gi kallende modul en tilbakemelding. Det kan man gi

1. i form av en Exception
2. som boolean (true=alt gikk greit, false=noe gikk galt)
3. som et tall for hvor mange objekter som ble skapt
4. en kombinasjon

Løsningen med boolean gir lite informasjon. Klart bedre er derfor et antall kombinert med feilmelding. Da kan man angi et antall skapte objekter, evt. 0 hvis alt gikk greit men filen var tom, og Exception ellers:

```
public int lesFil (String filnavn) throws Exception {
     BufferedReader innLeser=null;
     int antSkapt=0;
     try {
          innLeser
                  = new BufferedReader(new FileReader(filnavn));
          String linje = innLeser.readLine();
          while (linje!=null){
               //TODO:
               //splitt linjen og skap objekt som legges i en samling
               antSkapt++;
               linje=innLeser.readLine();
          }
     }
     //Behøver ikke catch - skal kaste feilen uansett
     finally {
         try {
             innLeser.close();
         }
         catch (Exception ex) {
             //Bare fang denne så evt. opprinnelig feil returneres
         }
     }
     return antSkapt;
}
```

Legg merke til et par triks her:

1. *innLeser* lukkes uansett i *finally*.
2. Jeg behøver ikke fange feilen i hoveddelen hvis jeg ikke skal skape en ny og annen *Exception*
3. En evt. feil i *finally* fanges da den ellers vil overstyre den opprinnelige feilen som kanskje oppsto.

Et råd er å **aldri returnere eller avslutte i *finally***.

Pass altså på å gi kallende modul en eller annen tilbakemelding når noe kan gå galt. Lukk alltid filer i *finally*. Det er en typisk bruk av *finally*.

Metoder som kan returnere uten resultat

Bakgrunn

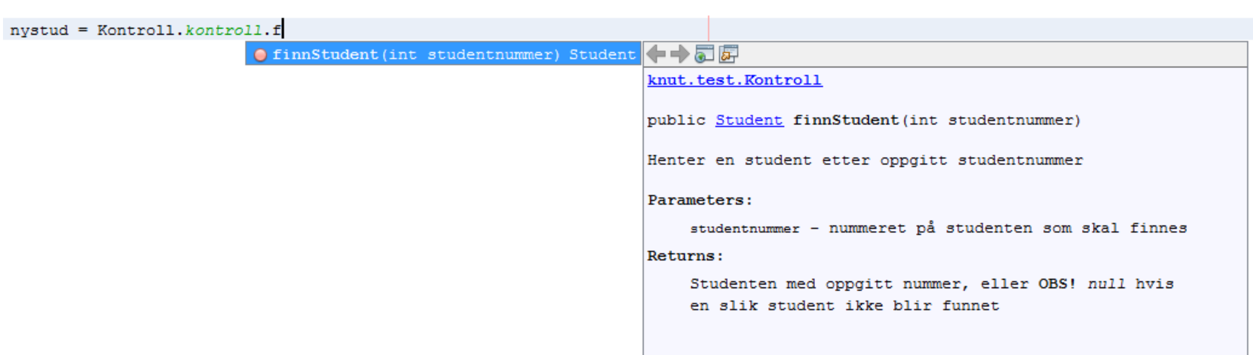
Noen metoder kan resultere uten resultat, f.eks. fordi det ble søkt etter noe som ikke finnes. Anta f.eks. at det finnes en metode

```
Student finnStudent(int studentnr)
```

Metoden går igjennom en samling med Student-objekter. Hvis studenten med oppgitt studentnr finnes, returneres Student-objektet. Hvis studentene *ikke* finnes kan man da tradisjonelt

1. Returnere *null*
2. Kaste *unchecked exception*, f.eks. *IllegalArgumentException* og *NoSuchElementException*
3. Kaste *checked exception*, f.eks. *Exception*

Ulempen med teknikk 1 og 2 er at metoden ikke behøver/kan merkes så programmereren som lager kallet ser det. Man bør legge inn godt med dokumentasjon, men allikevel kan det lett bli oversett:



Både metode 2 og 3 kan også sies å være feil bruk av *exceptions* siden det å ikke finne en bestemt student i registeret neppe kan sies å være en *feil* – det er mer en helt normal hendelse. Det skulle helle i retning av teknikk 1 og den er nok den mest brukte.

Alternativ fra Java 8: Returnere et Optional-objekt

Fra og med Java 8 kan man velge et annet – og mer naturlig alternativ – nemlig å returnere et objekt av klassen *Optional<T>*. Det kan inneholde et objekt av type *T* eller være tomt.

I eksemplet deklarerer man da metoden *Optional<Student>*. Det vil du se og måtte forholde deg til når du skal kalle metoden. Metoden som kanskje finner en student kan da se slik ut:

```
public Optional<Student> finnStudent(int studentnummer) {
for (Student s : studenter) {
if (s.getNr() == studentnummer) {
return Optional.of(s);
  }
}
return Optional.empty();
}
```

Her kan det også passe å bruke en strøm kombinert med Lambda-uttrykk som returnerer en *Optional<Student>* direkte:

```
public Optional<Student> finnStudent(int studentnummer) {
    return studenter.stream()
            .filter(s -> s.getNr() == studentnummer).findFirst();
}
```

findFirst() returnerer et objekt av typen *Optional<Object>*. Hvis du prøver å kalle denne metoden for å tilordne et *Student*-objekt, vil du få en slik feilmelding:

```
Student nystud;
String navn;

nystud = Kontroll.kontroll.finnStudent(10); //en student som finnes
```

incompatible types: Optional<Student> cannot be converted to Student
----
(Alt-Enter shows hints)

Du må altså deklarere *nystud* som *Optional<Student>* og sjansen er da svært stor for at programmet vil ta høyde for at metoden *finnStudent* ikke returnerer noe. Det kan du f.eks. gjøre slik (merk funksjonen *isPresent* som er true/false og *get* som evt. henter Student-objektet):

```
Optional<Student> nystud;
String navn;
nystud = Kontroll.kontroll.finnStudent(10);
if (nystud.isPresent()) {
navn = nystud.get().getNavn();
}
else {
JOptionPane.showMessageDialog
(rootPane, «Studenten finnes ikke registrert»);
}
```

Merk at *nystud* ikke er et Student-objekt, men et Optional-objekt. Hvis du glemmer deg og skriver *nystud.getNavn()* gis derfor feilmelding:

cannot find symbol
symbol: method getNavn()
location: variable nystud of type Optional<Student>

(Alt-Enter shows hints)

```
72
73
74
75
76
77
            navn = nystud.getNavn();
```

Det kan synes som du må gjøre mye ekstra når du kaller en metode som returnerer *Optional*, men husk da at du ellers enten måtte håndtere evt. *null* eller en exception. Mengden av kode blir omtrent den samme.

Ekstra: Optional som parameter

Java har ikke *optional* parametre. Klassen Optional<T> gjør allikevel slike parametre mulig. Her er f.eks. en konstruktør som setter nytt navn bare hvis det er oppgitt:

```
public Student(int nr, Optional<String> navn){
    this.nr = nr;
    if (navn.isPresent()){
        this.navn = navn.get(); //henter strengen
    }
}
```

Denne konstruktøren gjør ingenting med *navn* hvis det ikke er oppgitt. Hvis du heller vil ha en default-verdi – her f.eks. en tom streng – kan du bruke *orElse* slik:

```
public Student(int nr, Optional<String> navn){
    this.nr = nr;
    this.navn = navn.orElse(""); //returnerer enten strengverdien eller
""
}
```

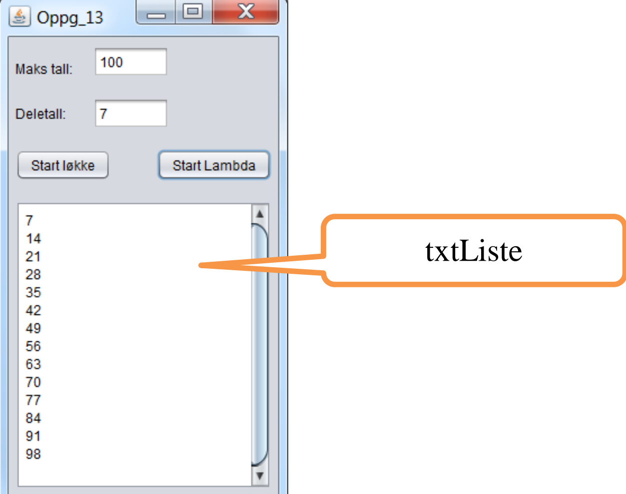
Ulempen er at en slik metode også må kalles med et *Optional* argument:

```
Student stud_1 = new Student(10,Optional.empty());
Student stud_2 = new Student(15,Optional.of("Knut"));
```

Oppgave til kapittel 13

Her skal vi anvende funksjonelt grensesnitt, en strøm og et Lambda-uttrykk. Grensesnittet skal testes.

Problemstillingen er at vi vil ha skrevet ut alle tall som er delelig med et annet opptil en grense, f.eks. alle tall som er delelig med 7 fra 1 opp til 1000, dvs. {7,14,21,...994}. Begge tall er heltall. Skjemaet kan se slik ut:



Klassediagram:

Det skal lages et funksjonelt grensesnitt *Filtrering* med default-funksjonen *filtrer*. Den returnerer true hvis *tall* er delelig med *deletall*.

Kontrollklassen implementerer dette grensesnittet så metoden *filtrer* blir tilgjengelig der. Det er en singleton som realiseres med to metoder. Strengene som returneres er på en form som kan settes direkte inn i skjemaets *txtListe*.

Kontrollklassens metode *løkke* går igjennom alle tallene og avgjør – med funksjonen filtrer fra grensesnittet – om tallet er delelig og i så fall tas det med i returstrengen (husk også linjeskift).

Kontrollklassens metode *lambda* bygger først opp en *LinkedList* med alle tallene. Denne listen omgjøres så til en strøm som filtreres. Bare de elementene i listen som slipper igjennom filteret, tas med i returstrengen.

Oppgave A:
Lag systemet

Oppgave B:
Gjennomfør enhetstesting av grensesnittet *Filtrering* med JUnit og rett opp koden om nødvendig.

Kapittel 14 – Ekstra fagstoff for egen lesning

I dette kapittelet tar jeg for meg forskjellig fagstoff som ikke er omtalt andre steder og som bør være av interesse for en dyktig programmerer. Stoffet kan leses uten forelesning.

Synlighet i Java V14

Kilde: http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html (noe endret av meg)

Access Levels					
Modifier	**Class itself**	**Classes in same package (subclass or not)**	**Subclasses outside package**	**World**	**UML**
public	Y	Y	Y	Y	+
protected	Y	Y	Y	N	#
no modifier	Y	Y	N	N	~
private	Y	N	N	N	–

Legg spesielt merke til betydningen av *private*. Man kan lett tro at det betyr at kun objektet selv kan se medlemmet, men faktisk kan også *alle andre objekter av samme klasse* se det.

Min anbefaling:

1. Bruk *private* for
    a. alle felt unntatt konstanter. Vær oppmerksom på at hvis feltet refererer til et *objekt* så er det *referansen* til objektet som ikke kan endres, mens objektet selv allikevel kan endres hvis det har metoder som gjør det mulig.
    b. metoder som bare objektet selv skal kunne se, f.eks. hjelpemetoder, hjelpefelt og set-metoder for felt som ikke andre skal kunne endre (f.eks. endring av identifikatoren).
2. Bruk *protected* for medlemmer som alle i pakken skal se, og alle subklasser uansett pakke.
3. Bruk *public* for medlemmer som alle skal kunne se. Det er vanlig for metoder som ikke endrer objektet (f.eks. get-metoder).
4. Bruke ingen synlighet for medlemmer som bare objekter i pakken skal se.
5. Bestem alltid bevisst en synlighet for alle felt. Det er svært sjelden bruk for ikke å angi synlighet, da subklasser vanligvis skal ha tilgang uansett pakke. Hvis du ikke angir noen synlighet bør du kommentere det så man ser at du har tenkt på det, f.eks. /*package*/String getNavn(){...
6. Tenk også over om selve klassen bør beskyttes, f.eks. med synlighet bare innen pakken. Det er sjelden aktuelt, det normale er *public* men det forekommer for "hjelpeklasser" til kontrollklassen o.l.

Java Regular Expressions (regex)

java.util.regex

Regular Expressions (regex) er et uttrykk – et mønster – som kan brukes til å søke i og endre strenger. Det finnes for mange programmeringsspråk, herunder Java. Regex brukes også i databasene MySQL og Oracle.

De systemene som ikke selv har regex implementert, kan vanligvis hente inn biblioteker som kan brukes.

Regex har litt varierende syntaks – Javasyntaksen finner du på http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Pattern og Match
For å skape et regex-uttrykk må du lage et Pattern-objekt. Pattern-klassen har ingen tilgjengelig konstruktør – du må bruke klassemetoden *compile*:
```
Pattern regex = Pattern.compile("[æøåÆØÅ0-9]");
```

Tilsvarende gjelder for Matcher som skapes av et Pattern-objekt:
```
Matcher match = regex.matcher("Kalvø Åsen");
```

Dermed har du regex-uttrykket knyttet til teksten som det skal søkes i. Da kan du liste ut alle forekomstene (*vis* er en textarea):
```
while (match.find()){
    funnet = true;
    vis.append(match.group() + " ble funnet f.o.m.  tegn nr "
            + match.start() + "\n");
}
```

Videre kan du erstatte tegn/deltekster som matcher med andre tegn/deltekster:
```
vis.append("Med de angitt tegn erstattet med §: "
     + match.replaceAll("§"));
```

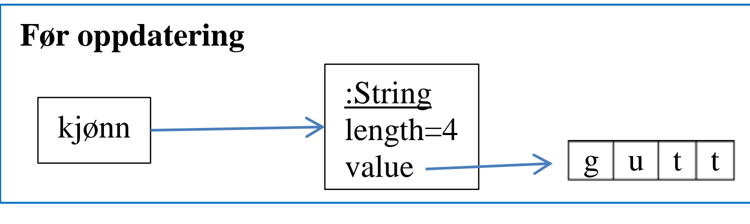
Til erstatning kan man egentlig like enkelt bruke string-metoden *replaceAll(String regex, String replacement)* som gjør nøyaktig det samme:
```
vis.append("Kalvø Åsen".replaceAll("[æøåÆØÅ0-9]", "§"));
```

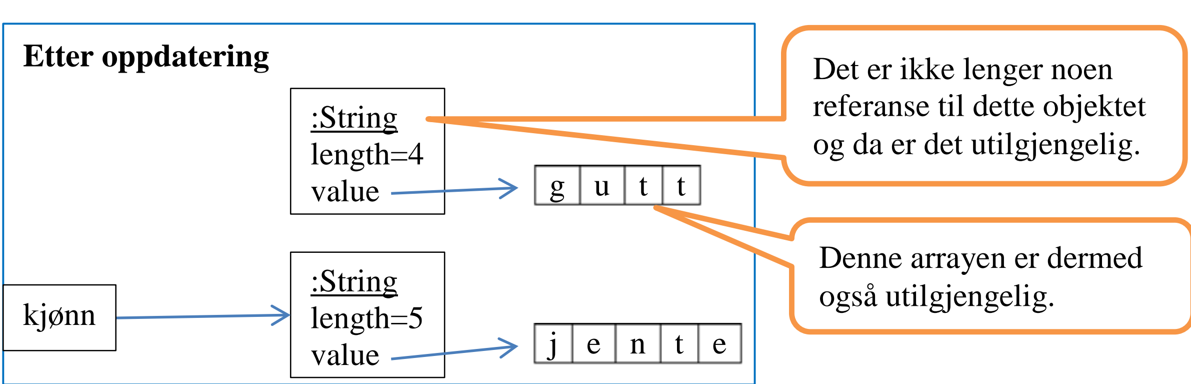
Bibliotek for regex-uttrykk
Hvis du leter etter et uttrykk for å teste om noe er OK, f.eks. en epost, en URL, en dato, så kan jeg anbefale http://www.regxlib.com/ Den er søkbar og har et meget stort antall mønstre.

Om StringBuilder og StringBuffer

String
Et problem med *String* er at den er immutable. Det innebærer at hver gang den endrer verdi så må strengen opprettes på nytt sted i Heap. Senere må *Garbage Collector* rydde opp i den gamle strengen.

Eksempel:
```
String kjønn = "gutt";
kjønn = "jente";
```





For en enkel tilordning, betyr denne overheaden ingenting, men hvis oppdateringen skjer i en iterasjon, kan det forsinke fordi det blir ganske mye overhead. Hver tilordning fører til at runtimesystemet må finne ny plass i RAM som er tilstrekkelig stor og opprette et nytt objekt der. Det tidligere objektet med sine verdier må *Garbage Collector* før eller senere ta seg av.

StringBuilder

StringBuilder er mutable. Det innebærer at den kan endre verdi, uten å endre plassering i RAM. Det ordnes ved at når en *StringBuilder* skapes, settes det straks av plass til et antall tegn (for tiden er standard 16 tegn, men JavaDoc lover ikke noe bestemt størrelse).

Man benytter metoder for å legge til flere tegn – *insert* eller *append*.

Hvis det blir for mange tegn, utvides plassen og *da* må et nytt *StringBuilder*-objekt skapes med tilsvarende overhead som for String. Det kan derfor lønne seg å sette av tilstrekkelig plass med én gang ved å oppgi et antall tegn i konstruktøren, f.eks.

```
StringBuilder utstreng = new StringBuilder(50);
utstreng.append("gutt");
utstreng.append(" og ");
utstreng.append("jente");
System.out.println(utstreng.toString());
```

Den siste setningen krever litt overhead ved at *StringBuilder*-objektet må konverteres til en streng, men det går svært raskt og krever ingen omplassering i RAM.

StringBuilder er ikke synkronisert og egner seg best for oppdatering av bare én tråd.

StringBuffer

StringBuffer har akkurat samme funksjonalitet og virker på samme måte som StringBuilder, men den er synkronisert. Det er en fordel når flere tråder oppdaterer samme streng, men krever ekstra overhead og er derfor litt tregere.

Litt kodeteori fra Internett

På nettet diskuteres ofte programmering. Ofte er eksemplene lit artige, men allikevel med en seriøs undertone. Det er ofte svært erfarne programmerere som forteller/drøfter. Nedenfor har jeg samlet et lite knippe, med svært forskjellig tema og utgangspunkt.

1. Dalip Mahal: "The Programmer Productivity Paradox"

Mahal begynner med å begrunne hvorfor programmerere bør kunne skrive minst 1000 kodelinjer pr måned (det blir ca 50 pr. arbeidsdag og burde vel ikke være umulig). Han fortsetter slik:

> Capers Jones has compared many **methodologies** (RUP, XP, Agile, Waterfall, etc) and **programming languages** over thousands of projects and determined that programmers write between **325** and **750** lines of code (LOC) per month, which is less than the 1,000 LOC per month suggested above. Even if programmers do not average 50 lines of code per day, the following is clear:
>
> - Methodology does not explain the apparent productivity gap
> - No language accounts for the apparent productivity gap
>
> The reality is that only a **fraction** of a developer's time is actually spent writing production code. If a developer is typing in code all the time then they are really trying different combinations of code until they finally find the combination of code that works. Or more correctly, the **combination** that seems to match the requirements until either QA or the business analyst comes back and lets them know there is a problem.

> That is why developers that **plan** their code before using the keyboard tend to outperform other developers. Not only do only a few developers really plan out their code before coding but also years of experience do not teach developers to learn to plan. In fact studies over **40 years** show that developer productivity does not change with years of experience.

Jeg har særlig sans for det siste siterte avsnittet om behovet for å planlegge koden før kodingen. Et annet interessant faktum er at det er meget stor forskjell på gode og dårlige programmerere – se No Experience Required! – problemet er at erfarne programmerere tydeligvis ikke lærer det!

2. David Green: Are Comments Always Wrong?

Det er vanskelig å skrive "passe" med kommentarer. Her er en litt humoristisk artikkel om det.

> A colleague asked me recently:
> Why aren't developers writing comments any more?
>
> He'd been looking through some code his team had written, and couldn't understand it – he was looking for comments to make sense of the mess, but there were none. Before he challenged the team, he asked my opinion: should developers be writing comments?

Excessive Comments

> When I started programming some years ago, I would comment *everything*, and I mean *everything*.
>
> ```
> // Add four to x
> x += 4;
> ```
>
> My logic at the time was that it was impossible to tell the difference between uncommented clear code and gnarly code you just hadn't spotted the gnarliness in yet. So I would comment everything, where the absence of a comment meant I'd forgotten – not that it was so trivial as to not warrant mentioning.

No Comments

> Eventually I started working with peers who knocked some sense into me, and I immediately halved the number of lines of code I produced. At first it was a shock, but soon I realised that clear code is easier to read *if there's just less noise*. And this is all (most) comments become: noise. Sometimes they're accurate, sometimes they're not. But you can't *rely* on them, you always have to assume they might be wrong. So if the code and the comment seem at odds, you assume it's the comment that's wrong and not your understanding of the code (naturally you're infallible!)

Clean Code

> Uncle Bob and the notion of clean code have taken "no comments" to an almost fanatical zeal. But every time I get into an argument with someone about how maybe this time a comment might be justified: Ctrl-Alt-M, enter your comment as the method name and it makes the code more obvious. Every. Damned. Time.
>
> However, the trouble with a zealous argument like this is it gets taken up by asshats and lazy people. It's too easy to say "if you'd read Clean Coder you'd know you don't need comments. Quit living in the past, grandpa!". Uh huh. But your code is still a muddled pile of indecipherable crap. At least with comments there'd be some signposts while I wade through your steaming mess.

Some Comments

> The truth is: sometimes comments **do** help (squeal clean code weenies, squeal!) There are some cases where extracting a method name *isn't sufficient*. Sometimes having 20 one line methods in my class does not make it easier to read. The end goal is to produce *understandable* code. Generally, naming things properly helps. Adding comments that get stale does not. But that doesn't mean that writing crap code *and* not commenting is the answer. Don't use "no comments" as an excuse to leave your code indecipherable by human beings.
>
> For example, sometimes you need to document why you *didn't* do something. Maybe there's a standard way of converting between currencies in your application – which this one time you've deliberately not used. A comment here might help future people not refactor away your deliberate choice (even better is baking your decision into a design – some class structure that makes it *really* obvious). Sometimes a method name really doesn't do a line of code justice, it's better to be seen in the context of the lines before and after it – but it really needs some explanation of what you're doing. This is particularly true when dealing with complex algorithms or mathematical formulae.

Getting the Balance Right

> How do you get the balance right? Well, your goal is to make code that other people can understand. The best way to get feedback on that is to *ask* someone. Either have explicit code reviews or pair program. If another human being has read your code and *they* could understand it – there's a better than average chance that most other capable people will be able to read it too.

Prinsippet "clean code" innebærer at det ikke skal skrives kommentarer. En viktig begrunnelse er at når koden endres, har ingen tid til å endre kommentarene. De blir derfor etter hvert bare til forvirring. "Clean code" stiller store krav til koden.

Siste avsnitt gir et godt råd: Finn ut hva som er passe med kommentarer ved å vise koden din til andre.

3. Lars Marius Garshol

Lars er norsk og har en logg med mye forskjellig. Jeg for min del har stor sans for hans råd om bra kode – se http://www.garshol.priv.no/blog/105.html. Det som står der, er råd jeg selv stadig gir studentene mine, så her fant jeg en "soul mate".

4. Antonin Januska: What Fibonacci taught me about programming

Fibonacci-tallene har vi lekt med flere ganger. Det er et bra eksempel på rekursive funksjoner. Denne artikkelen er imidlertid om noe annet, nemlig effektiviteten av kode. Figuren som viser hvordan rekursiv beregning av Fibonacci-tallene sprer seg utover i RAM er virkelig instruktiv – og skremmende. Januska har faktisk kjørt benchmark for forskjellige løsninger.

(Koden er uten innrykk av tekniske grunner – han skriver nok ikke koden slik. Ja, og la deg ikke skremme av matematikken – du behøver ikke forstå den for å ta poenget!)

> I originally wanted to write this neat piece on how solving the "get a number from the Fibonacci sequence" problem can teach YOU a lot but over the past year, I've realized that programming is such a personal experience that I believe I'd do you injustice by preaching and schooling you.
>
> Instead, I'd like to focus on my own experience with it.

A few weeks ago, I was solving the Fibonacci sequence problem:

Given an index, return the correct number from the Fibonacci sequence.

I tried different approaches, looked up how others have done it and learn a few awesome things.

Recursion is badass but can be misleading

One solution that I've seen was the recursive one. Recursion to me has always been magic. Partially because you have to approach a problem with a certain mindset to be able to use recursion. And I rarely ever find problems that require that mindset.

Anyways, so I found a solution and just looking at it, I thought that it was the best thing ever.

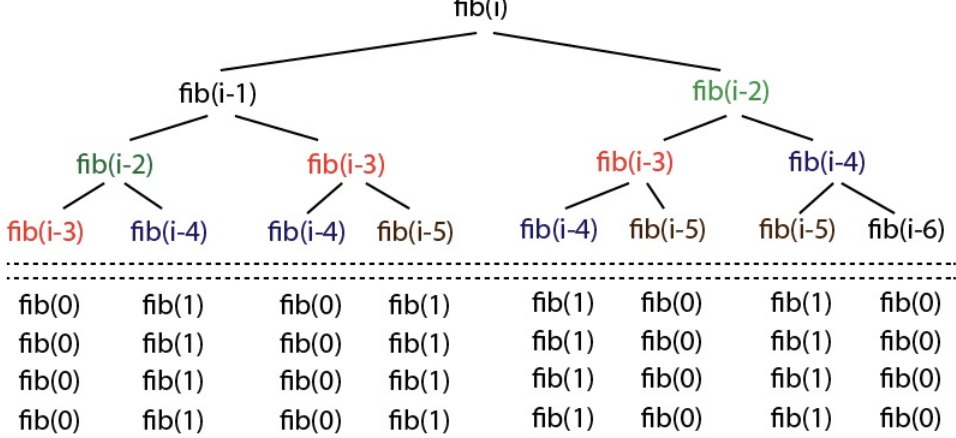
First, let me show you the functional (mathematically functional) representation of Fibonacci:

1.F(n) = F(n-1) + F(n-2)

Basically, each Fibonacci number is equal to the sum of its two predecessors. The recursive solution looks very much like the mathematical representation:

```
1.function fibRec(i) {
2.if (i < 2) {
3.return i;
4.} else {
5.return ( fibRec(i-1) + fibRec(i-2));
6.}
7.}
```

I thought to myself, "This is perfect! So simple, and it describes itself! This MUST be the perfect solution!" but it's not. Look at what happens when you run this neat recursive function:



As you can see, the function gets recursively worse in terms of performance. We're exponentially calculating the same equation. I'd call this the "near-infinite loop". Because we calculate and recalculate the same problem over and over and over. The larger our i gets, the more exponentially worse the recursion gets.

So I reminded myself of an old saying:

just because it looks right, it doesn't mean it is.

I learned that not every problem that looks like a recursion-solvable should be solved with recursion. Technically, this is a prime example of where recursion mirrors the nature of the problem but is not a good solution.

Then comes solving programmatically (or Naive)

Every problem out there has an inelegant "naive" solution. I say "naive" because it's seemingly inefficient. For instance, the solution to Fibonacci using plain programming is this:

```
01.//programmatically
02.function fibProg(i) {
03.var a = 0;
04.var b = 0;
05.var c = 0;
06.
07.for(var d = 0; d < i+1; d++) {
08.if(d === 1) {
09.c = 1;
10.} else {
11.c = a+b;
12.}
13.a = b;
14.b = c;
15.}
16.
17.return c;
18.}
```

As you can see, we have to count up from 0 all the way up to the index every single time. It's terribly inefficient. Isn't there a way to just "skip" to the correct number instead of this iterative approach? No, not with just programming. However, the solution is performant enough that you can leave it at that. No need to go further.

The recursion solution could result in maximum call stack errors, overflows, and what have you. This solution won't and it will work efficiently up to some arbitrary high number.

The Mathematical Solution

Being a former math major, I remembered that there is a Fibonacci solving formula based on the golden ratio. A quick look up in Wikipedia resulted in a sweetly simple equation:

$$F_n = \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n .$$

I decided to plug it into a function and create a fibMath function for a solution:

```
01.function fibMath(i) {
02.var sqrtFive = Math.sqrt(5);
03.var firstHalf = 0;
04.var secondHalf = 0;
05.
06.firstHalf = 1 / sqrtFive * Math.pow( ( ( 1 + sqrtFive ) / 2), i);
07.secondHalf = 1 / sqrtFive * Math.pow( ( (1 - sqrtFive ) / 2 ), i);
08.return Math.round(firstHalf - secondHalf);
09.}
```

It's a pretty neat but completely un-understandable function for a programmer. However, the performance is staggering.

I love how math can cut through problems like butter through innovative formulas. I was taken aback and realized that there are things about programming on the fundamental, mathematical level that I may not fully understand but allow us to work at the speeds we work at.

Note: Someone pointed out to me that the Math.round() function will eventually end up with incorrect solutions due to the floating point error and similar problems. There are some workarounds if you need to be precise and will be using this function out in the real world.

The Tests

I decided to test all of the solutions with JSPerf, a cool little app that employs Benchmark.js for benchmarking.

I set up as fair of a test as I could and put all three solutions to work.

First up is a benchmark that allows fibRec (the recursive solution) to be used. I found that the maximum call error pops up in the range of 50 or above (perhaps even lower). So I started with getting the 20th index.

Check it out. Here are the results for my computer:

	Test	Ops/sec
Recursive solution	`fibRec(index);`	6,443 ±1.18% 100% slower
Programmatical Solution	`fibProg(index);`	15,239,521 ±1.30% 50% slower
Mathematical	`fibMath(index);`	30,482,504 ±0.94% fastest

To further test the programmatical vs mathematical solutions, I picked a higher index to compute in a "hardcore mode".

Check that out and see the difference:

Testing in Chrome 33.0.1750.146 on OS X 10.9.2		
	Test	Ops/sec
Programmatical solution	`fibProg(index);`	2,006 ±0.97% 100% slower
Mathematical Solution	`fibMath(index);`	4,700,770 ±6.94% fastest

Note that this hardcore mode tests for the index of 54320 instead of 20. What's interesting is that while the mathematical solution is much MORE performant, the programmatical solution works just fine. Unless I'm directly dealing with math solutions, there's no need to go nuclear and convert functions into mathematical representations (although, that would be a neat project).

I believe that game programming relies on mathematical formulas due to these difference, however, they do actually run these equations several million times a second.

So I learned about optimization

The difference between the programming solution and the mathematical solution reminds me of the difference of running native assembly code and something like PHP. The math solution is like magic, and so is assembly (sometimes). But it's not practical in most cases, in terms of debugging, understanding, and changing.

The entire process taught me that it's important to avoid potential bottlenecks like the first solution but it's just as important to not get hung-up at the performance of solutions like the programming solution. I thought it was going to be slow but I was wrong. At 2000 ops/sec with a very high look up index, there was nothing to worry about whatsoever. That's a performant solution.

Could I do better? Yeah. And I did, the mathematical solution gave me performance several times over the programming one. And if I was using Fibonacci or other mathematical concepts out in the real world, I would have learned my lesson to use the smart way and not the "manual" way (manual, as in, the computer manually computes a solution rather than cutting straight through to the answer).

The downside to the math solution is that it's not often available for most real-world problems (outside of video gaming and 3D). The non-math solution is good enough though and should not be dismissed.

On the other hand, a less performant but understandable/maintanable solution is much better than a very performant yet illegible is a trap that should be avoided.

I don't know if that makes sense, but it was an interesting exercise.

Min konklusjon: Rekursjon er bra, men iterasjon er gjerne mer effektivt. Hvis man kan tenke seg litt om før man løser problemet, blir det enda bedre.

Om kryptisk kode

Enkelte dyktige programmerere hevder at kode skal være enkel, da er det større sannsynlighet for at den er korrekt. Og den blir enklere å teste.

Her er et eksempel fra en eksamensbesvarelse som ikke akkurat følger ovenstående prinsipp:

```
kontrollObj.finnEiendom(Integer.parseInt(JOptionPane.showInputDialog(
"Eiendomsnummer")))
.setVerditakst(Integer.parseInt(JOptionPane.showInputDialog("Verditak
st)));
```

Denne er ikke enkel å forstå! Er den riktig? Hvordan går det f.eks. om den feiler et eller annet sted, f.eks. i *parseInt*? Hva slags feilmelding gis da? Og hva om *finnEiendom* ikke returnerer noen eiendom men *null*?

Denne synes jeg bør deles opp:

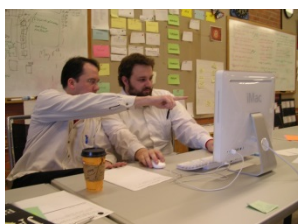
1. Be om et eiendomsnummer og kontroller det. Er det tomt? Klikket brukeren et tall? Evt. gi feilmelding.
2. Gjør om nummeret til heltall. Fang feil og gi evt. feilmelding.
3. Kjør finnEiendom og kontroller resultatet. Returnerte den en eiendom? Evt. feilmelding.
4. Be om verditakst og kontroller det. Er det et tall? Gi evt. feilmelding.
5. Gjør om verditaksten til et tall. Fang feil og gi evt. feilmelding.
6. Kjør setVerditakst og kontroller returen (den kaster kanskje feil?). Evt. gi feilmelding.

Da blir det enklere å forstå og betydelig enklere å teste!

Pair Programming

Beskrivelse

"Pair Programming"[53] er en teknikk som er mye brukt i "Agile Software Development"[54]. Det dreier seg om at to programmerere samarbeider tett med å programmere på én maskin. Den ene – kalt "driver" (her kalt "sjåføren") – er "i førersetet" og har kontroll over tastaturet. Sjåføren skriver koden. Den andre – kalt observer, pointer eller navigator (her bruker jeg ordet observatør) – følger nøye med.



Sjåføren konsentrerer seg om selv kodingen. Observatøren kontrollerer at koden blir korrekt mens den skrives, og har også ansvaret for å se om det kan være andre måter å gjøre det på, om forbedringer kan gjøre koden mer robust og/eller vedlikeholdsvennlig. På denne måten får sjåføren konsentrert seg om å skrive kode, mens observatøren er et sikkerhetsnett.

Sjåfør og observatør bytter plass jevnlig etter en viss tid eller når en liten arbeidsoppgave er ferdig.

Dette krever naturligvis stor åpenhet og teamfølelse. De to får jo intimt kjennskap til hverandres kompetansenivå. Det er viktig at arbeidet skjer som et tett team og ikke ender i konkurranse om å være best, konflikter e.l. På den annen side er det motiverende å jobbe så tett med en annen – programmering beskrives ellers ofte som "ensomt".

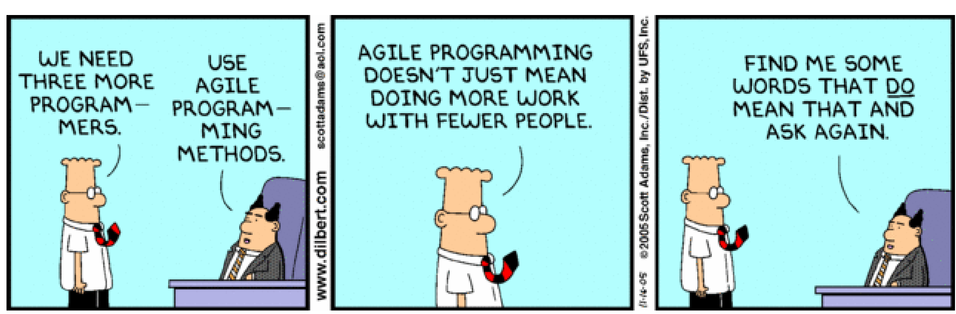
---

[53] Jeg har ikke sett noen god oversettelse av dette til norsk. Dette kan kanskje oversettes til "programmering i to-spann" eller noe slikt? I praksis brukes det engelske uttrykket.

[54] Forsøksvis blir det oversatt til "smidig systemutvikling" og settes i motsats til "fossefall systemutvikling". I praksis brukes "Agile" også på norsk.

Kan dette lønne seg?
I Programming Pair settes jo to personer til å lage én kode. Man skulle tro at det ville være mer effektivt at de to kodet hver for seg parallelt. Det har vært gjort en rekke studier av dette. De konkluderer noe forskjellig. En metastudie[55] konkluderte slik:

> *This study suggests that even though coding is often completed faster than when one programmer works alone, the total amount of man-hours (the product of the number of programmers and the time spent) increases. A manager would have to balance faster completion of the work and reduced testing and debugging time against the higher cost of coding. The relative weight of these factors can vary by project and task. The benefit of pairing is greatest on tasks that the programmers do not fully understand before they begin: that is, challenging tasks that call for creativity and sophistication. On simple tasks, which the pair already fully understands, pairing results in a net drop in productivity. Productivity can also drop when novice–novice pairing is used without sufficient availability of a mentor to coach them.*



Metastudiens hovedkonklusjon var at fordelene ved Pair Programming kunne være at oppgaven ble løst raskere (mindre tid) og at det kunne bli høyere kvalitet, men fordi to programmerere brukes på én oppgave, går effektiviteten ned (økte arbeidstimer). Dette så ut til å variere med situasjonen, f.eks. betydningen av å bli ferdig fort og økt kvalitet mot økte kostnader. Fordelene ser ut til å bli størst når programmererne skal løse noe de ikke fullt forstår på forhånd – en typisk situasjon for studenter og nybegynnere.

Uansett synes jeg det ville være en fordel om dere studenter fikk prøve denne måten å jobbe på.

Exceptions – en diskusjon

To grupper Exceptions
I Java er det to grupper Exceptions – checked og unchecked.

Unchecked Exceptions oppstår pga logiske programmeringsfeil (bugs). Man kan ikke regne med at brukeren eller programmet skal kunne rette opp feilen. Programmet kan like godt stoppes. Slike feil arver fra klassen *RuntimeException*. Metoder kan kaste slike feil uten at metoden *må* merkes med det og uten at de som bruker metoden *må* håndtere feilen. Faktisk hevdes det å være vanligst at feilen ikke håndteres (det er vanskelig å håndtere den også).

Checked Exceptions oppstår pga forhold utenfor programmererens kontroll, f.eks. pga feil brukerinput, databasefeil, filhåndteringsfeil eller nettverksproblemer. Dette er feil som programmereren kan ta høyde for og sørge for at de blir rettet, evt. med brukerens hjelp.

---

[55] En metastudie tar for seg mange undersøkelser som er gjort og prøver å trekke en endelig konklusjon fra deres resultater. De oppnår altså en slags samlet konklusjon uten å gjøre eksperimenter selv.

Programmet kan gjerne fortsette å kjøre etterpå. Slike feil arver fra klassen *Exception*[56]. Metoder som kaster slike feil *må* enten håndtere dem selv, eller så *må* de merkes og da *må* de som bruker metoden håndtere dem (eller merkes tilsvarende).

Problemer med Exceptions

Exceptions – enten de er checked eller ikke og enten de håndteres eller ikke, skaper flere problemer.

1. Ressurser som ikke frigis

Når en feil oppstår, avbrytes den vanlige programflyten. Det kan føre til at ressurser ikke blir frigitt. Det kan være en fil/database/connection osv. som blir stående åpen eller RAM (f.eks. et buffer, en datastruktur) som ikke blir frigitt. Transaksjoner kan også bli avbrutt uten tilbakerulling så dataene blir inkonsistente.

Dette lar seg løse, men det er ikke alltid like enkelt.

2. Bortkastet CPU

Hvis man litt tankeløst lager en try-catch rundt en algoritme som tar lang tid og som *kan* gå feil, risikerer man at det tar lang tid for så til slutt å feile. Det kaster bort verdifull tid.

Hvis f.eks. Telenor (noe tåpelig) leser samtlige kunder fra en database og lager objekt av hver enkelt, risikerer de at RAM går full eller at noe annet galt skjer før alle er på plass. Da er mye CPU-tid og ventetid for brukeren bortkastet.

Det kan løses ved å sjekke *før* man kjører den tunge algoritmen. I Telenors tilfelle f.eks. ved en enkel *select count(*)* og beregning av plassbehov, før *select **.

Generelt kan man løse det ved å forsikre seg om at det vil gå bra, *før* man kjører den tunge algoritmen.

3. Exceptions kan skjule bugs

Man kan løse mange bugs ved å lage en try-catch rundt, særlig hvis man bare fanger feilen uten å gjøre noe med den. Det fjerner naturligvis ikke noen bugs men bare skjuler symptomet. Da kan programmet feile et annet sted eller – enda verre – gi galt resultat.

Løses ved å finne bugs – ikke skjule dem.

4. Skjuler andre feil

En exception kan gi uthopp i flere nivåer, hvoretter eksekveringen fortsetter. Det kan skjule andre feil (som aldri blir funnet pga uthopp videre) og når tilslutt programmet virkelig stopper kan det bli svært vanskelig å finne hvor feilen *egentlig* oppsto.

5. Problemer med videreutvikling

Checked Exceptions er en del av metodens synlig grensesnitt. Hvis den f.eks. opprinnelig deklarerte visse typer av feil (si type A og B) og man legger til en ny kontroll som kaster C, så vil andre moduler feile fordi grensesnittet er endret – de er jo ikke programmert til å håndtere feil type C.

Man skal være svært forsiktig med å endre en metodes grensesnitt på et senere tidspunkt i systemets livssyklus fordi det vil påvirke alle andre metoder som anvender den. Det er vanskelig å finne alle metoder som anvender en annen bestemt metode.

Løsning med å deklarere alle som den generell *throws Exception* er ikke bra – feilene er forskjellige så det følger forskjellige data med og de har forskjellige metoder.

---

[56] For å forvirre litt så arver faktisk også klasse *RuntimeException* fra *Exception*, men den overstyrer altså oppførselen mht krav om håndtering.

6. Skalerbarhet
I små systemer kan alt virke greit. Hvis du kaller én API som kan gi fire forskjellige feil, så er alt vel, men hvis den kaller to nye som igjen kaller fem, så blir det svært mange forskjellige feil som skal håndteres – alle havner tilslutt i din kode som må håndtere dem alle.

7. Hvor skjedde feilen?
Hvis du har en sekvens på 10 setninger som alle kan feil, bør du egentlig ha separat try-catch på hver av dem. Det gir mye ekstra kode og et rotete program.

Alle som skal bruke metoden må håndtere alle disse feilene – ofte på forskjellig måte.

8. Kaste en annen type feil
Noen kan fristes til å fange en type feil og skape en ny og annen type som kastes (fra catch). Da mister man all informasjon om stack som viser nøyaktig hvor den opprinnelige feilen oppsto.

Det er altså å regne som "bad practice".

9. Manglende logging
Å kaste en feil kan hindre at en feil og årsaken til den blir logget. Det er vanlig å logge alle Exceptions, men da må mye tas med i loggen. Hvis feilen bare kastes, blir loggen lite meningsfull. En feil logget som "IllegalArgumentException in lesFil()" gir lite info om hvorfor feilen oppsto. Hva var argumentet? Hvorfor var det feil?

Man må huske at try-catch skal virke også i produksjon – ikke bare under testing.

10. Try med tom catch
Dette regnes av en rekke fagfolk som en "dødssynd" eller bent fram som en bug. Feil som fanges skal håndteres der og da eller overlates til kallende metode som kan håndtere dem (eller sende dem videre oppover til en som kan).

11. Åpner opp implementeringen
Det er mange gode grunner til å skjule implementeringen. Typer av Exceptions som deklareres åpent, ødelegger en del av denne skjulingen.

F.eks. vil en deklarasjon av *findUser (String user) throws SQLException* tydelig vise at dataene hentes fra en database, noe som kanskje ingen andre har noe med. Feilen som kastes er da på feil nivå. Man burde isteden kaste en feil kalt *UserNotFound* eller liknende. Evt. kunne man returnere *null* (i pakt med "radikalt syn" nedenfor).

Fordeler med Exceptions
Exceptions er ikke uten fordeler ellers hadde de vel neppe blitt brukt så meget.

1. Reliabilitet
Det kan være høyst aktuelt at programmet skal unngå krasj. I en atomreaktor som plutselig "opplever" noe svært uvanlig, vil vi neppe at styringsprogrammet skal stoppe. Heller ikke i et fly. Og vi kjenner vel alle problemet med "blåskjerm" når Windows gir opp helt. Feilen logges men er ikke enkel å finne[57]. Jeg har nylig opplevd slike fordi noen ikke fanget feil i Outlook når serveren uventet gikk ned.

2. Enkel løsning
I programmer som skal kjøres bare én gang (laget for et bestemt formål) kan ha stor glede av en enkel, kjapp try-catch. Feilen er hvis de brukes i mer seriøse sammenhenger også. Finn feilen – ikke skjul den med try-catch!

[57] Programmet "BlueScreenView" kan kanskje hjelpe.

3. Dekker "skumle" kall
Når man kaller API som man ikke stoler helt på, vil man nødig at den andre koden feiler og trekker ens egen med i dragsuget. En try-catch rundt slike "skumle" kall kan beskytte din egen kode.

4. Håndterer sjeldne hendelser
Ofte kan man sjekke hendelser selv *før* man utfører noe. Det er god praksis. Hvis det er mye som skal sjekkes, kan koden bli svært tung og mye av sjekkingen er lite nødvendig fordi feilen antas å opptre sjelden.

F.eks. er det fornuftig å sjekke at et oppgitt filnavn ikke er null eller en tom streng og kanskje at filen finnes. Når en fil skal åpnes og skrives til er det imidlertid så meget annet som kan gå galt. Hvis man skal sjekke for alt dette, blir koden unødvendig lang og mye av kontrollen vil det aldri bli bruk for (katalogen finnes ikke, stasjonen er ikke montert og tilgjengelig, disken er full, manglende rettigheter, filen er opptatt eller skrivebeskyttet osv.).

5. Gir en tilbakemelding uansett
Funksjoner skal gi en returverdi. Hvis det ikke er mulig – av en eller annen grunn – er det tross alt bedre å få en feil. Den kan da sees på som en retur og skal ideelt sett gi informasjon om hvorfor det ikke var mulig. Hvis feilen er checked antar man at kallende metode kan gjøre noe med det, evt. sammen med brukeren, ellers bør feilen være unchecked.

Det krever naturligvis at feilen ikke "svelges" av en tom catch.

Radikalt syn
Det finnes fagfolk som mener at Checked Exceptions generelt er en dårlig idé. En slik er Biswas som argumenter godt på http://deathbycode.blogspot.no/2010/12/problems-with-checked-exception-in-java.html.

Andre mener at det beste vil være at absolutt alle metoder gir en tilbakemelding i form av en verdi. Noen ganger er det enkelt, f.eks. hvis en funksjon skal returnere alder så kan den returnere -1 som tegn på at noe gikk galt. Andre ganger er det vanskelig/umulig å finne en slik "ulovlig" verdi og da er det verre. VBs *tryParse* løser det ved å returnere *to* verdier – et variabelt argument endres til resultatverdien mens funksjonsverdien forteller om parsingen gikk greit. En annen løsning kan være å returnere et objekt med to verdier – den ene er resultatet den andre feilen (evt. null).

For min del synes jeg man bør prøve å sjekke argumentene før man starter. Jeg kaster da gjerne en Checked Exception hvis argumentet er feil, fordi det ofte er noe som en bruker har oppgitt og kan rette opp. Jeg er nok litt for glad i å bruke try-catch også når catch faktisk ikke kan rette feilen. Da hadde det nok vært bedre å la feilen returnere til den kallende metoden. Jeg er i alle fall av den mening at det blir for primitivt å bruke try-catch bare ved filbehandling. Man bruker mange ferdige objekters/klassers metoder som kan kaste feil og da bør man alltid tenke igjennom om man vil håndtere feilene, enten i en catch eller med en finally-gren.

Annen bruk av try-catch

I .NET har vi funksjonen `isNumeric(s As String)` As Boolean som er svært kjekk. Noe tilsvarende har ikke Java. Tidligere har dere sett en litt "sleip" bruk av try-catch for å oppnå det samme. Slik kan den se ut for heltall:

```
/**
 * Funksjon som sjekker om en streng kan konverteres til et heltall
 * med int Integer.parseInt(String s)
 * @param s En streng som kanskje kan omgjøres til et heltall
 * @return true hvis s kan omgjøres, ellers false
 * @author Knut etter ide i StackOverflow
 */
public boolean isInteger(String s){
    try {
        Integer.parseInt(s);
    }
    catch (Exception ex) { //hvis den feiler er s ikke et heltall
        return false;
    }
    return true; //feilet ikke, altså er s et heltall
}
```

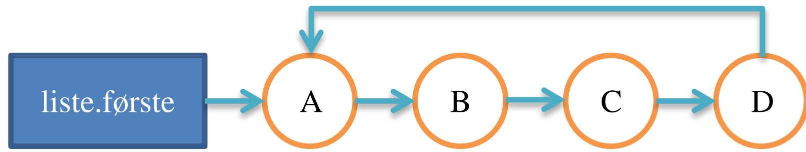
Hvis man bruker *double Double.parseDouble(String s)* på samme måten, får man sjekket om strengen *s* i det hele tatt er et tall.

```
/**
 * Funksjon som sjekker om en streng kan konverteres til en double
 * med int Double.parseDouble(String s)
 * @param s En streng som kanskje kan omgjøres til en double
 * @return true hvis s kan omgjøres, ellers false
 * @author Knut etter ide i Stack Overflow
 */
public boolean isNumeric (String s){
    try {
        Double.parseDouble(s);
    }
    catch (Exception ex) { //hvis den feiler er s ikke et heltall
        return false;
    }
    return true; //feilet ikke, altså er s et tall
}
```

Tilsvarende kan vi enkelt lage funksjoner som sjekker for byte, short, long og float.

Utfordring 14A til kapittel 14: Er listen rekursiv?

Man kan tenke seg at en lenket liste – ved en feil – er blitt rekursiv, f.eks. slik:



Hvis man blar i en slik liste vil man få en evig iterasjon og aldri finne slutten.

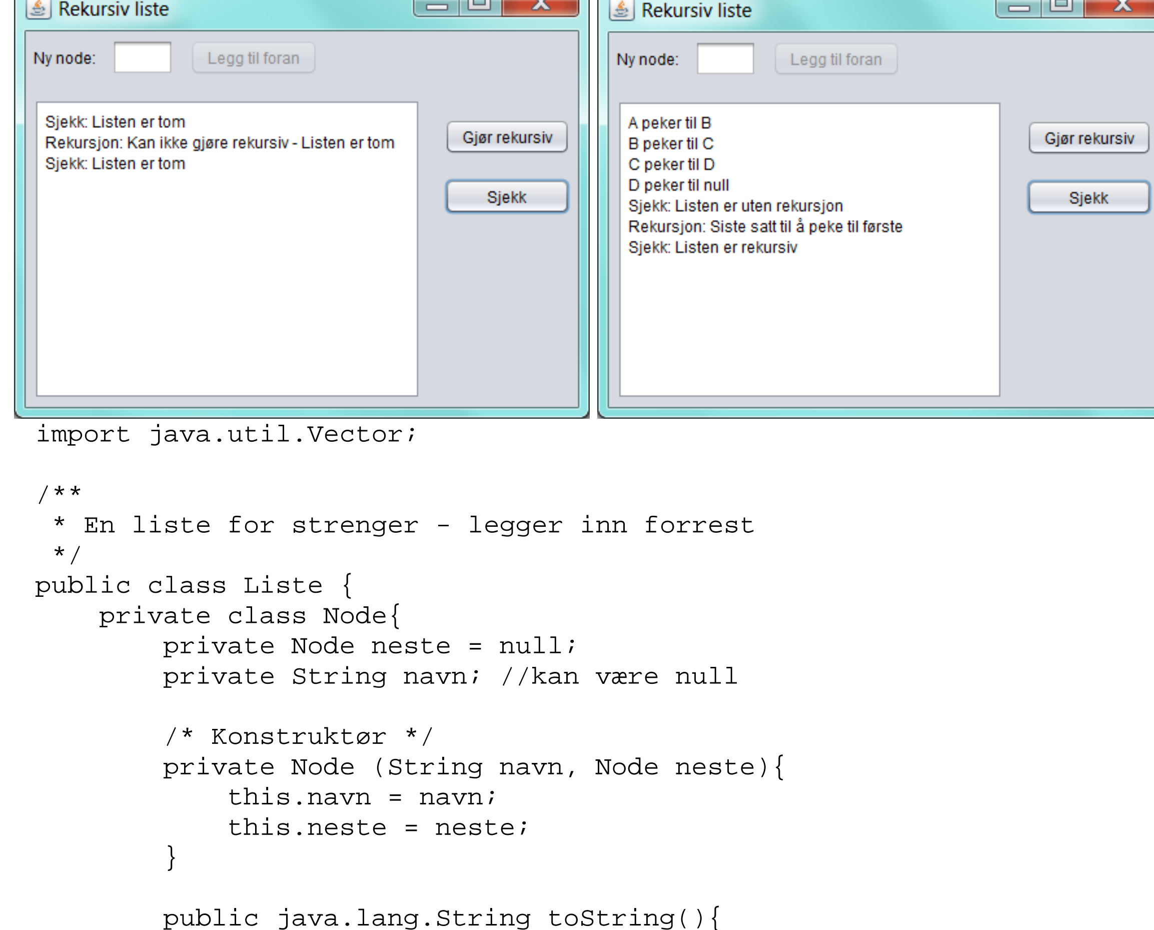
Oppgaven går ut på å sjekke om listen er rekursiv.

Vink

Du kan ikke bruke de ferdige ADT-ene for de kan ikke bli slik. Du må lage din egen liste.

På neste side finner du en mulig kode for Liste-klassen. Listen tar bare imot strenger. Jeg har fjernet de delene som gir utfordring. Skap en ny klasse *Liste* og lim inn koden herfra.

Mitt grensesnitt ser slik ut (først med en tom liste og deretter med listen ovenfor):

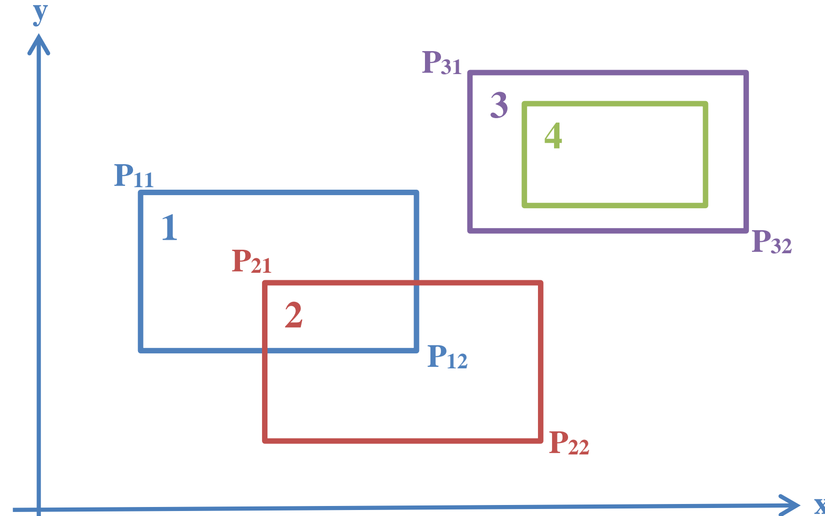



```
import java.util.Vector;

/**
 * En liste for strenger - legger inn forrest
 */
public class Liste {
    private class Node{
        private Node neste = null;
        private String navn; //kan være null

        /* Konstruktør */
        private Node (String navn, Node neste){
            this.navn = navn;
            this.neste = neste;
        }

        public java.lang.String toString(){
            return navn + " peker til " + (neste==null? "null" :
neste.navn);
        }
    }

    private Node første = null;

    /**
     * Legg nytt navn forrest i listen (kan være null)
     */
    public void add(String navn){
        //TODO: Legg til kode som plasserer den nye noden først
        }

    /**
     * Sett siste nodes peker til første
        //Finn siste node
        //Sett siste til å peke til første
        return "Ikke implementert";
    }
```

```
        /**
         * Hent alle noder i listen - bruker nodens toString
         * @return Vector<String> som kan være tom
         */
        Vector<String> alle(){
            Vector<String> tmp = new Vector<>();
            // TODO: Gjør ferdig
            return tmp;
        }

        /**
         * Kontrollerer om listen er rekursiv
         * Returnerer en streng som rapport
         */
         public String sjekk(){
            //TODO: Gjør ferdig
            return "Ikke implementert";
         }
    }
```

Utfordring 14 B til kapittel 14: Overlapper to rektangler?

Ett rektangel er gitt med to punkter: Øvre venstre punkt og nedre høyre punkt.



I figuren er rektangel 1 gitt ved punktene P_{11} og P_{12}, rektangel 2 ved punktene P_{21} og P_{22} osv. Hvert punkt har en x-verdi og en y-verdi. Som det fremgår overlapper rektangel 2 og rektangel 1 hverandre. Rektangel 3 overlapper hverken 1 eller 2. Rektangel 3 og 4 overlapper hverandre.

Første del:
Gitt to slike rektangler – beskriv kriteriene for overlapp, altså: Hvordan kan du bestemme om to slike rektangler overlapper?

Andre del:
Skriv programmet. Lag gjerne en klasse *Rektangel* gitt ved to punkter *Point* og en funksjon

```
boolean overlapp(Rektangel rekt1, Rektangel rekt2)
```

som returnerer sant hvis de to rektanglene overlapper hverandre.

Happy hacking!

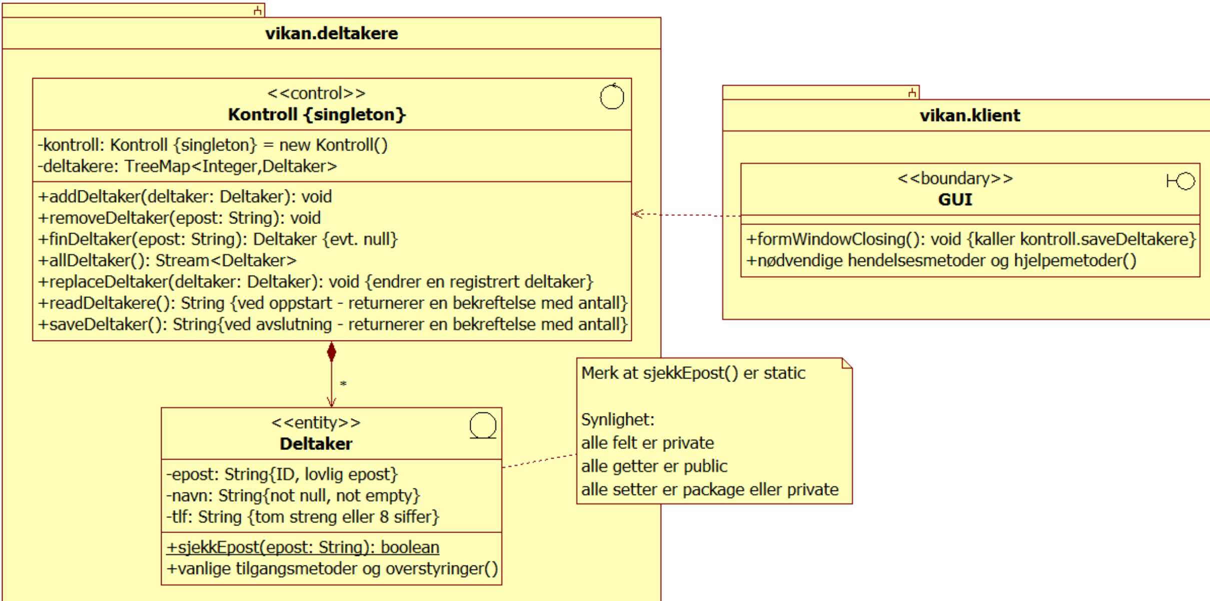
Avsluttende case – heldags prosjektoppgave

Situasjonsbeskrivelse

Du har søkt jobb i programvarefirmaet "ViKan" som lager programmer på oppdrag. Du er nå innkalt til en test. Der får du i oppgave å lage en prototype for et system. Du skal gjøre arbeidet alene.

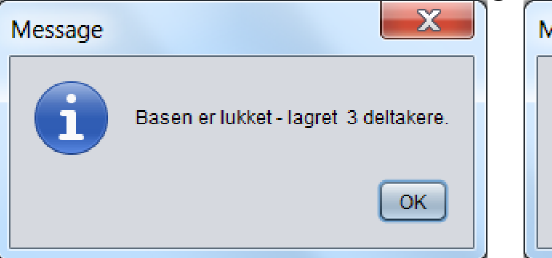
Applikasjonen

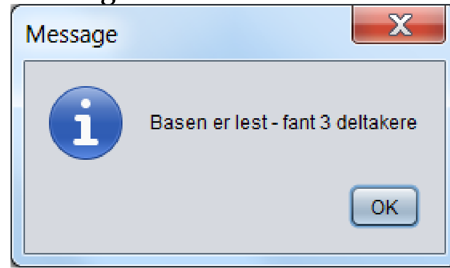
Du skal lage en prototype med *NetBeans* for en applikasjon som registrerer konferansedeltakere. ViKan har laget denne modellen, som du skal følge:



Legg merke til at klassene er fordelt på to pakker, da den endelige versjonen tenkes delt på klient/tjener. Merk også at *allDeltaker* kan returnere en tom *Stream* og at *findDeltaker* kan returnere *null*.

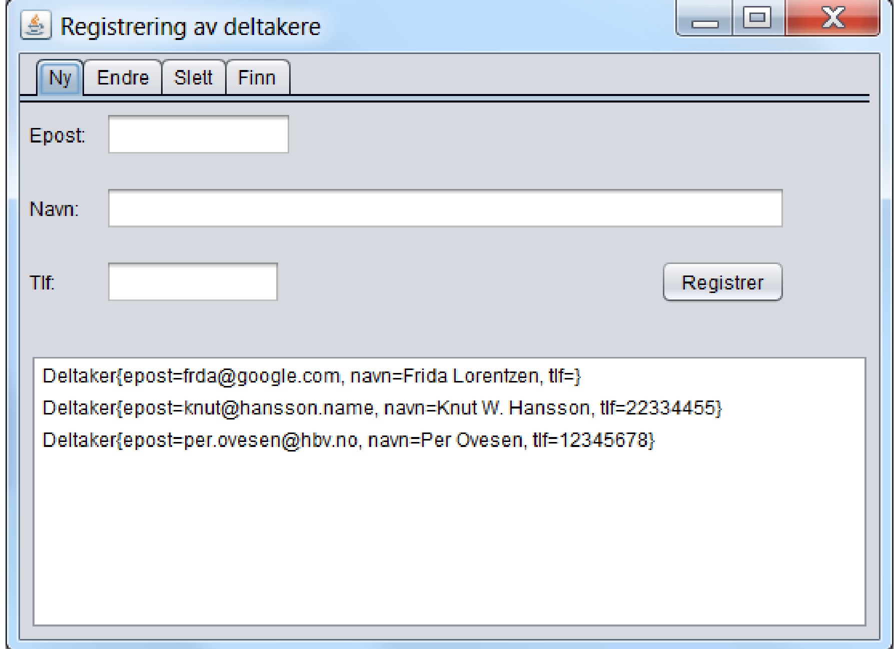
Når programmet starter, kaller GUI kontrollobjektets metode *readDeltaker* som henter alle deltakerne fra databasen til *TreeMap deltakere* og returnerer en streng som kan vises i en meldingsboks. Når GUI avslutter kaller det kontrollobjekts metode *saveAnsatte* som lagrer alle ansatte[58] og returnerer en tilsvarende streng. Meldingsboksene kan f.eks. se slik ut:





[58] *Vink:* Tøm tabellen *deltaker* helt og lagre alle pånytt.

Grensesnittet for GUI kan se slik ut:



De andre tre fanene ser temmelig like ut. Listen nederst synes uansett hvilken fane som er aktivert[59].

Alt skal lagres i Derby databasen *db_deltakere* i tabellen *deltaker*. Databasen har brukeren *data* med passord *d@ta*. Du må selv skape databasen.

Du kan regne med at alle objektene får plass i RAM.

Krav

a) All kode skal selvsagt kommenteres etter behov.
b) Kontrollklassen skal i tillegg dokumenteres med JavaDoc.
c) Klassen deltaker skal testes med JUnit og testklassen leveres sammen med det øvrige prosjektet.
d) Alle metodene i pakken *vikan.deltakere* kan kaste feil og klassen *GUI* skal håndtere dem[60].
e) Det kreves kontroll av domenene i klassen *Deltaker* slik de fremgår av klassediagrammet[61].
f) Kontrollklassens *public* API bør ikke endres, men ellers kan du fritt legge til metoder og felt som du har brukt for.

Vink: Du anbefales å arbeide i denne rekkefølgen: Lag *Deltaker*, test den, lag *Kontroll* uten databasehåndtering (med *readDeltakere* og *saveDeltakere* som stubber), deretter *GUI* fullfør databasehåndteringen i *Kontroll* og dokumenter så *Kontroll* med JavaDoc.

---

[59] *Vink:* lag en *JTabbedPane* øverst og en *JList* under den.
[60] *Vink:* De fleste (alle?) feil kan programmeres med en feilmelding som *GUI* kan formidle direkte videre til brukeren i en meldingsboks.
[61] *Vink:* Du finner regex-uttrykk for kontroll av eposter på http://RegExLib.com – se f.eks. Francesco DAguanno.