INF1010 v17: Tema-liste

## Enhetstesting

* Idéen er å teste de minste bestanddelene av programmet individuelt, og når man vet at de virker, kan man begynne å teste større deler av programmet.
* Hvordan lage gode test-cases
  + Test *edge cases*, altså de tilfellene som akkurat ikke skal gå eller akkurat skal gå

Bokhylle bokhylle = new Bokhylle(100);

Bok etDukkehjem = new Bok("Et dukkehjem")

// sjekk "venstre" side av arrayet

bokhylle.settInn(etDukkehjem, -1); // skal ikke gå

bokhylle.settInn(etDukkehjem, 0); // skal gå

// sjekk "høyre" side av arrayet

bokhylle.settInn(etDukkehjem, 99); // skal gå

bokhylle.settInn(etDukkehjem, 100); // skal ikke gå

* + - F.eks. skal det ikke gå å sette inn en bok på plass -1, men 0 skal gå. Og tilsvarende skal det gå an å sette inn på plass 99, men ikke 100.
* Prøv å teste færrest mulig ting samtidig for å gjøre det lettere å finne ut hva som feiler. F.eks. bør en metode gjørDittOgDatt deles opp i gjørDitt og gjørDatt, så de to tingene kan testes separat.

## Arv

* Enkle subklasser (uten polymorfi)
  + Subklassen er en spesialisering av superklassen
  + Klassehierarkier oppgave
    - <http://folk.uio.no/inf1010/v17/oppgaver/#ex:virveldyr-og-pattedyr>
* Nøkkelordet protected og tilgang til variabler
* Referanser

class A {}

class B extends A {}

* + Variabel av type A kan referere til objekt av B, men ikke omvendt
  + <http://folk.uio.no/inf1010/v17/oppgaver/#ex:arv-teori>
  + <http://folk.uio.no/inf1010/v17/oppgaver/#ex:arv-kodeanalyse>
* Datastrukturtegning av objekter av subklasser
  + Egentlig ikke noe hokuspokus. Objektet inneholder instansvariable definert i selve klassen og superklassen(e).
    - Man kan tegne datastrukturen fra superklassen og klassen selv ligger i to forskjellige deler av objektet. Se side 12 i [lysarkene fra forelesningen](http://www.uio.no/studier/emner/matnat/ifi/INF1010/v17/lysark/subkl1-2017.pdf).
* Nøkkelordet instanceof
  + o instanceof Student sjekker om objektet (som variabelen o refererer til) er av type Student
* Klassen Object
  + Alle klassers mor
    - Når vi definerer en klasse uten å skrive extends, vil klassen istedet bli en subklasse Object.
  + Har en del metoder som det er verdt å se litt på
    - toString()
      * Denne kalles når man prøver å printe et objekt.
    - equals(Object o)
      * Brukes for å avgjøre om to objekter er like. == sjekker om to referanser er til samme objekt, men equals kan sjekke om objektene er like i den forstand at hvis man hadde byttet dem om, ville det ikke hatt noen effekt.
      * Kan minne om at equals kan brukes for å sjekke om to strenger er like.
    - Eksempel på bruk av equals(Object o):

public class Test {

public static void main(String[] args) {

String a = new String("hei");

String b = new String("hei");

System.out.println(a == b); // gir false

System.out.println(a.equals(b)); // git true

}

}

* + - Eksempel på overskriving av toString() og equals(Object o):

class Bok {

private String tittel;

private String forfatter;

public Bok(String tittel, String forfatter) {

this.tittel = tittel;

this.forfatter = forfatter;

}

public String hentTittel() {

return tittel;

}

public String hentForfatter() {

return forfatter;

}

// Overskriver equals i Object

public boolean equals(Bok other) {

return tittel.equals(other.hentTittel()) && forfatter.equals(other.hentForfatter());

}

// Overskriver toString i Object

public String toString() {

return forfatter + ": " + tittel;

}

}

class Program {

public static void main(String[] args) {

Bok bok1 = new Bok("Big Java, Late Objects", "Cay Horstmann");

Bok bok2 = new Bok("Big Java, Late Objects", "Cay Horstmann");

// Siden vi har overskrevet toString i bokklassen, vil dette se pent ut:

System.out.println("Pensumboka i INF1010 er " + bok1);

// Vi kan også teste equals-metoden vår:

System.out.println("bok1 == bok2: " + (bok1 == bok2));

System.out.println("bok1.equals(bok2): " + bok1.equals(bok2));

}

}

* Metodesignaturer
  + Definisjon av metodesignatur i Java:> Two of the components of a method declaration comprise the method signature — the method's name and the parameter types.

For eksempel vil denne metoden

public double calculateAnswer(double wingSpan, int numberOfEngines,

double length, double grossTons) {

//do the calculation here

}

ha metodesignatur

calculateAnswer(double, int, double, double)

* En klasse kan ha flere metoder med samme navn, så lenge de har forskjellig signatur, for kompilatoren kan avgjøre signaturen ved å se på metodekallet. Å avgjøre returtypen blir derimot mer problematisk fordi det er lovlig å kalle en metode uten å lagre eventuell returverdi. Det er også viktig å presisere at det er typene og rekkefølgen (ikke navnene) til parametrene som er av betydning.
* Polymorfi
  + Overriding
    - Subklasse redefinerer metode definert i (en av) superklassen(e).
  + Overloading
    - Metoder med samme metodenavn, men forskjellig *signatur*

## Grensesnitt og Arv

* Grensesnitt
* Arv med grensesnitt
* Introdusere lenkelister
  + Enkel lenkeliste
  + Fokus på tegning og hvordan datastrukturen ser ut
* Syntaksen til grensensnitt:
  + Deklarere:

public interface NavnPaaInterface(){

void metode();

int metode2(int a, int b);

}

* + Grensesnitt-metoder har ingen implementering
  + Grensesnittmetoder er alltid public (trenger derfor ikke å skrive public)
  + Interface kan ikke ha instansvariabler
  + Interface kan ikke ha statiske metoder

## Lenkeliste

* FIFO/LIFO
* Indre klasser
* Generiske klasser

public class Boks<T> {

private T t;

public void fyll(T t) {

this.t = t;

}

public T hent() {

return t;

}

* Har en variabel first/head som refererer til første elementet i listen. Fra den kan man få tak i resten.
* FIFO-liste - First in, first out. Ex. Kø. Setter inn nye elementer bakerst.
* LIFO-lista - Last in, first out. Ex. Stack. Setter inn nye elementer forrerst.
* Forskjellen på LIFO og FIFO i programmering er måten man setter inn nye elementer på.
* I stedet for at hver objekt har en egen neste-peker, lagrer man heller disse objektene i en Node-klasse, som er en private klasse innenfor lenkelisten. Nodens oppgave er å ha en neste-peker og en peker til objektet den holder på.
* Ved sletting av et element så må man oppdatere det forrige elementet sin neste-peker, til å peke på det elementet man slettet sin neste-peker.
* Nevn at det også er mulig å ha dobbelt-pekere, dvs. at lista både har en neste- og en forrige-peker i et Node-objekt.
* Invarianter: ta høyde for om listen er tom (feks før man prøver å fjerne element).

## Iterator

* Lenkelister og iteratorer \* Ikke bare lenkelister som kan bruke iterator
* Klassen som er "listen", den som skal iterere implements Iterable
* Iteratoren man lager implements Iterator
* Trenger ikke å lage filene Iterable.java og Iterator.java
* Interfacet til Iterable:

public interface Iterable<T>{

Iterator<T> iterator();

}

* Interfacet til Iterator:
  1. boolean hasNext()
  2. T next()
  3. void remove() dersom remove ikke skal implementeres kan man skrive ""

public interface Iterator<T>{

boolean hasNext();

T next();

void remove();

}

* Fordelen med iterator: gjør det mulig å feks bruke for-each-loop for å iterere gjennom alle elementene i en beholder.
* metoden compareTo() ligger allerede inne for feks int eller String.
* Forklare at compareTo() er en metode vi selv kan definere
* Relevante treningsoppgaver: - 6.1 Arraybasert liste med iterator - 3.2 Comparable

## Rekursjon

* Metode som kaller på seg selv

public int factorial(int n) {

if(n == 2) {

return 2;

}

return n\*factorial(n-1);

}

public int fibonacci(int n) {

if(n == 1) {

return 1;

}

if(n == 2) {

return 1;

}

return fibonacci(n-1) + fibonacci(n-2);

}

public static int finnStorsteInt(int[] array, int index,int storst) {

if(index == array.length) {

return storst;

}

if (array[index] > storst) {

storst = array[index];

}

return finnStorsteInt(array, index + 1, storst);

}

## Tråder

**Basics**

**Hva er en tråd?**

En tråd er en sekvens av instruksjoner i et program ("thread of execution"). Hittil har vi skrevet vanlig, ikke-parallelle programmer. Disse har hatt én sekvens av instruksjoner, altså én tråd.

Hvis vi ønsker at to eller flere oppgaver (sett med instrukser) skal kjøre *samtidig*, kan vi lage flere tråder. Disse oppgavene vil da kunne utføres i parallell.

Ulike tråder kan dele på objekter (minne), og "snakke" med hverandre. I INF1010 skal trådene ikke kommunisere direkte, kun bruke delte objekter.

**Thread = worker, Runnable = task**

Vi kan tenke på en tråd som en arbeider - noe som kan utføre et sett med instrukser, en oppgave. Arbeiderne er objekter av klassen Thread.

Oppgavene som trådene kan utføre er objekter av grensesnittet Runnable.

**Runnable**

Vi beskriver oppgaver som kan utføres ved objekter av grensesnittet Runnable:

interface Runnable {

void run();

}

**Eksempel:**

class MyTask implements Runnable {

/\* Say hello, but think for a while between each time. \*/

public void run() {

for (int i = 1; i < 5000; i++)

if (i % 1000 == 0)

System.out.printf("Hello for the %d-th time!\n", i / 1000);

}

}

**Thread**

Vi lager nye arbeidere ved å lage et objekt av klassen Thread. Vi kan så gi tråden en oppgave den kan utføre, og så be den starte.

**Eksempel:**

public Main {

public static void main(String[] args) {

Runnable task = new MyTask();

Thread worker1 = new Thread(task);

Thread worker2 = new Thread(task);

worker1.start(); // Will make a call to task.run()

worker2.start();

}

}

Output (eksempel, ikke alltid samme utskrift):

Hello for the 1-th time!

Hello for the 1-th time!

Hello for the 2-th time!

Hello for the 3-th time!

Hello for the 2-th time!

Hello for the 4-th time!

Hello for the 3-th time!

Hello for the 4-th time!

**Alternativ måte å lage tråder på:**

Vi kan også lage tråder ved å lage en subklasse av Thread, hvor vi lager run inne i trådobjektet:

class MyThread extends Thread {

@Override

public void run() {

...

}

}

class Main {

public static void main(String[] args) {

Thread worker = new MyThread();

worker.start();

}

}

Selv om dette er mulig, skal vi (stort sett) holde oss til å skille mellom tråder og oppgaver (Runnable). Dette er fordi vi ønsker å kunne gjenbruke *samme* tråd på *forskjellige* oppgaver, eller samme oppgaver utført av flere tråder. Lager vi tråder på måten vist her ender vi opp med *spesiallist-tråder*, som bare kan gjøre én ting. Det er også mer *objektorientert* å skille mellom oppgaver og arbeidere.

**Synkronisering**

Dersom to tråder har tilgang på delt data, kan vi få uheldige resultater dersom vi ikke passer på.

**Eksempel: Problemet**

public class Main {

public static void main(String[] args) {

Runnable task = new MyTask();

Thread worker1 = new Thread(task);

Thread worker2 = new Thread(task);

worker1.start();

worker2.start();

}

}

class MyTask implements Runnable {

private final int MAX\_COUNT = 10000;

private int sharedCounter = 0;

public void run() {

System.out.println("Starting! Shared counter = " + sharedCounter);

for (int i = 0; i < MAX\_COUNT; i++) {

sharedCounter = sharedCounter + 1;

}

System.out.println("Done! Shared counter = " + sharedCounter);

}

}

Forventet utskrift er at tråden som er ferdig sist skriver ut at telleren er 2 \* MAX\_COUNT

Starting! Shared counter = 0

Starting! Shared counter = 0

Done! Shared counter = 10297

Done! Shared counter = 11953

Problemet er at begge trådene forsøker å oppdatere samme verdi *samtidig*. Dette kalles en *race condition*, og er (kanskje) den største kilden til feil i parallelle programmer.

**Løsningen:**

Vi løser problemet ved å sørge for at kun en tråd kan aksessere et delt objekt til en gitt tid. Vi lager en såkalt *mutex*, en kritisk region hvor kun en tråd er om gangen. Andre tråder må stoppe og vente på sin tur.

**Locks**

Vi lager en mutex med *låser*. Tenk at du har dører på hver side av koden du vil beskytte. Hvis trådene låser døren når de går inn, må de andre trådene vente til tråden som låste låser opp igjen.

Vi bruker java.util.concurrent.locks.Lock, et grensesnitt som spesifiserer (blant annet) to metoder: lock() og unlock(). Den faktiske klassen vi lager låsobjekter av heter java.util.concurrent.locks.ReentrantLock.

**Eksempelet: Problem løst**

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

// Main uendret

public class Main {

public static void main(String[] args) {

Runnable task = new MyTask();

Thread worker1 = new Thread(task);

Thread worker2 = new Thread(task);

worker1.start(); // Will make a call to task.run()

worker2.start();

}

}

class MyTask implements Runnable {

private final Lock lock = new ReentrantLock();

private final int MAX\_COUNT = 10000;

private int sharedCounter = 0;

public void run() {

System.out.println("Starting! Shared counter = " + sharedCounter);

for (int i = 0; i < MAX\_COUNT; i++) {

lock.lock();

try {

sharedCounter = sharedCounter + 1;

} finally {

lock.unlock();

}

}

System.out.println("Done! Shared counter = " + sharedCounter);

}

}

Output:

Starting! Shared counter = 0

Starting! Shared counter = 0

Done! Shared counter = 16792

Done! Shared counter = 20000

Nå kan kun en og en tråd oppdatere telleren, så vi får ønsket oppførsel.

Legg merke til bruk av try{...} finally{lock.unlock();}. Dette sørger for at låsen blir låst opp, uansett hva som måtte gå galt mens låsen var låst. Dette er en sikkerhetsmekanisme for å unngå *deadlocks* (alle venter på låsen, ingen kan låse opp).

(Akkurat i dette eksempelet er det ikke mye som kan gå galt, og det ville trolig gått fint uten å bruke try/finally. Likevel lager vi oss gode vaner og gjør det *hver* gang vi bruker en lås. På den måten slipper vi uheldige feil senere.)

**Monitors**

En god strategi for å beskytte delt data er å lage en *monitor*. En monitor er et objekt som innkapsler den delte dataen, og definerer synkroniserte metoder for å jobbe med dataen. Vi skal i INF1010 *alltid* benytte monitorer, da dette er en god, objektorientert måte å gjøre synkronisering på.

**Eksempel: Problem løst, med monitor**

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class Main {

public static void main(String[] args) {

CountMonitor monitor = new CountMonitor();

Runnable task = new MyTask(monitor);

Thread worker1 = new Thread(task);

Thread worker2 = new Thread(task);

worker1.start();

worker2.start();

}

}

class CountMonitor {

private final Lock lock = new ReentrantLock();

private int sharedCounter = 0; // The protected data.

public void increment() {

lock.lock();

try {

sharedCounter = sharedCounter + 1;

} finally {

lock.unlock();

}

}

public int getCounter() { return sharedCounter; }

}

class MyTask implements Runnable {

private final int MAX\_COUNT = 10000;

private final CountMonitor monitor;

public MyTask(CountMonitor monitor) { this.monitor = monitor; }

public void run() {

for (int i = 0; i < MAX\_COUNT; i++) {

monitor.increment();

}

System.out.println("Done! Shared counter = " + monitor.getCounter());

}

}

Her ender vi opp med samme output. Gleden med å bruke en monitor vil bli større når problemene blir mer komplekse.

**Conditions**

Mange ganger må tråder vente på at en betingelse er oppfylt før den kan gjøre noe fornuftig. Dette kan vi oppnå ved å bruke java.util.concurrent.locks.Condition. En Condition kan vi tenke på som en ventekø tilknyttet en monitor. Vi kan plassere tråder i denne køen, i påvente av at en betingelse skal bli sann. Når dette skjer kan vi *signalisere* tråden, slik at den kan fortsette.

Condition er et grensesnitt som for oss har tre relevante metoder:

interface Condition {

void await();

void signal();

void signalAll();

}

**Barrierer**

En barriere tenkes på som et samlingspunkt i koden, der alle stopper og venter på at alle andre også har kommet frem. Deretter kan vi la alle trådene fortsette, *samtidig*.

* CountDownLatch(int count) - Alle slipper gjennom barrierer når .countDown() har blitt kalt count ganger. Vi kan *velge* å vente ved barrieren med .await().
* CyclicBarrier(int count) - Når count tråder har kalt på .await(), slipper alle gjennom. Barrieren kan så *brukes på nytt* på samme måte.

## GUI

* JavaFX
  + Application
  + Stage og Scene
  + Noder
  + EventHandler

**Application**

* Alle applikasjoner vi lager arver fra javafx.application.Application (vis gjerne Java-dokumentasjonssidene underveis)
* Vi @Overrider den abstrakte metoden start (Stage stage) med ønsket funksjonalitet for vår applikasjon
* Når vi skal kjøre applikasjonen kaller vi bare metoden launch() i main-metoden vår (launch arves fra Application).
  + Når vi launcher en Application oppretter vi samtidig en uavhengig applikasjonstråd.

**Stage**

* Når vi launcher en applikasjon opprettes det et Stage-objekt som blir sendt med som referanse og som vi kan bruke i vår definisjon av start(Stage stage).
* Stage er hovedbeholderen for GUI-et og inneholder blant annet informasjon om max størrelse på vinduet som skal vises frem, tittel og Scene-objektet vi vil vise.
* For å vise vinduet til brukeren må vi kalle på stage sin show()-metode.

**Scene**

* Scene-objektet inneholder blant annet et eller flere layout-objekter (GridPane, VBox osv.), og kan også registrere hendelser og styre "kameraet".
* Vi kan instansiere et Scene-objekt ferdig med layout, eller legge til dette senere.

**Noder**

* Kort sagt: Alt du putter i en Scene.
  + Layout (panes)
  + Button, Label, TextField osv
  + Kontroll av events
  + En del noder kan ha "barnenoder" - for eksempel kan man ofte "nøste" panes (dette må ikke forveksles med arv)
    - Vi legger til barn under en node ved å hente ut "barnelisten" med getChildren(), for deretter å kalle på listas add(nyNode) eller addAll(nyNode1, nyNode2, nyNode3, ...)

**Panes**

* Flere å velge mellom, avhengig av hva vi trenger (tegn gjerne opp eksempler underveis).
* VBox/HBox: Legger alt i en vertikal eller horisontal linje
* GridPane: Lager et rutenett
* StackPane: Gjør det enkelt å legge elementer oppå hverandre fra bakerst til forrest
* Se dokumentasjon for flere alternativer

**EventHandler**

* Grensesnitt vi bruker for å bestemme hva som skjer når vi registrerer en hendelse
* Eksempel: setOnAction(EventHandler eventHandler), for eksempel i Button og TextField

**Generelt om GUI**

* Det å skrive GUI i Java krever litt research - ting forandrer seg og gjøres kanskje på andre måter enn vi er vant med til nå
  + Eksempelvis: Bruk av anonyme klasser - dette må dere ta stilling til om dere vil bruke, men husk at det ikke er påkrevet. Lag tradisjonelle klasser hvis dere er mer komfortable med det.
* Som alltid viktig å planlegge programmet i forkant.
  + Planlegg layout ved hjelp av skisser
  + Tenk over hva slags struktur programmet bør ha for at ting skal fungere optimalt. Kan du holde GUI og logikk adskilt?
  + Grafikkprogrammering i Java krever relativt mye import av biblioteker. Dersom dere er komfortable med det kan det derfor være greit å bruke et IDE som letter denne jobben, evt finne en plugin til favoritt-teksteditoren din.
    - Det er ikke det å huske allverdens JavaFX-biblioteker som blir viktig på eksamen.

**Scene graph**

* Når vi jobber med GUI blir disse nodene en del av applikasjonens scene graph. Scenegrafen kan vi tenke på som et hierarki, eller mer spesifikt et tre av noder.
* Treet består av branch nodes (grener) og leaf nodes (bladnoder).
* Branch nodes er noder som kan ha andre noder under seg i hierarkiet. Disse arver fra klassen Parent. Dette gjelder for eksempel alle Pane-klasser (GridPane, HBox osv) og Group (en klasse som, i motsetning til panes, ikke gir noen layout til elementene den inneholder).
  + Noder som arver fra Parent inneholder en liste med barnenoder. Denne kan vi hente ut og deretter modifisere, for eksempel ved hjelp av getChildren().addAll(Node ...)
  + Den "øverste" noden kaller vi en root node (rotnode), og det er denne vi legger til i Scene-objektet. En hvilken som helst Parent kan være rotnode.
* Leaf nodes må, i motsetning til branch nodes, være nederst i hierarkiet, dvs de kan ikke ha barnenoder. Dette gjelder for eksempel Shape-klasser (Circle, Square ...) og Canvas.
* Igjen: Viktig at dette ikke forveksles med arv!

**Styling**

* Innebygde metoder
* CSS
  + Tidligere benyttet man seg typisk av innebygde metoder for å stilisere GUI-elementer. Senere har standarden blitt å bruke CSS, som gir litt flere muligheter.
  + Kort for de som ikke har erfaring med CSS:
    - Relativt enkelt språk som brukes for å beskrive presentasjon av blant annet HTML
    - Kan for eksempel brukes for å beskrive farger, dekorasjoner, størrelser ...
  + I Java kan vi hovedsakelig bruke CSS på to måter
    - Egne dokumenter med CSS-beskrivelser (.css)
    - Inline (dette kommer vi innom i eksempelet) som argumenter i koden vår
  + Klasser som arver fra Node arver også metoden setStyle(String value)
    - Dette kan vi bruke til å gi noden en CSS-stil

**Lytting på andre elementer enn knapper**

* Vi kan lytte på diverse noder på omtrent samme måte som med knapper. For eksempel kan vi lytte på interaksjon med former (rektangler, sirkler osv).
* Hvis vi ønsker å lytte etter klikk bruker vi metoden setOnMouseClicked() som finnes i alle Node-subklasser.
* I tillegg kan vi for eksempel lytte etter om museknappen holdes nede eller slippes, eller om vi holder nede knappen og drar (for eksempel i et tegneprogram).

**Laste inn filer**

* Vi kan laste inn en eller flere filer ved hjelp klassen FileChooser sine metoder showOpenDialog(Window owner) eller showOpenMultipleDialog(Window owner). På denne måten får brukeren en dialogboks med tilgang til et filtre.
  + Vi sender med "eier" av dialogboksen, som regel nåværende Stage-objekt. Da blir bakgrunnsvinduet blokkert frem til vi har gjort oss ferdig i dialogboksen.
  + Vi kan også legge til filtre for å begrense hva slags filer brukeren kan velge (*live-kodes i Gallery.java*)
* Vi kan også hente en utfil fra brukeren ved hjelp av FileChoosers showSaveDialog(Owner owner).
* Skulle vi trenge å hente en hel mappe kan vi istedet bruke klassen DirectoryChooser.

**Implementere GUI i eksisterende program**

* Flere måter å gjøre det på:
  + La hovedprogrammet arve fra Applikasjon og kalle launch(args) i main-metoden (som regel det letteste)
  + Skrive GUI-applikasjonen i separat klasse og launche den fra en hovedklasse (for de interesserte). Eksempel:

//Applikasjon

public class TestApp extends Application {

@Override

public void start(Stage stage) {

//GUI-arbeid her

}

}

//Hovedprogram

public class Hovedprogram {

public static void main(String[] args) {

Application.launch(TestApp.class, args);

}

}