Prøveeksamen 2020

Hvordan ville jeg ha løst den?

For innspill og spørsmål underveis, ta kontakt på chat med Siri Moe Jensen eller Stein Gjessing.

Dag Langmyhr (dag@ifi.uio.no)

Eksamen

Oppgaven består av to typiske deler:

1. **Modellering**
   Gitt et scenario, hvilke klasser og interfacer trenger vi for å kunne representere det i et oo-program.
2. **Datastrukturer og programmering**
   Velge riktig datastruktur til et problem, og så programmere det hele.

Prøveeksamen 2020: Oppgave 1

Emballasjefabrikken *Renpakk* skal lage et nytt datasystem (i Java) for å administrere sine produkter, og du har fått i oppdrag å lage deler av dette systemet.

Alt Renpakk produserer kalles Emballasje, og det er fire hovedtyper: Glassemballasje, Metallemballasje, Plastemballasje og Pappemballasje.

Noe av emballasjen til Renpakk er det pant på, og noe emballasje er nedbrytbar. Dette er egenskaper som kommer i tillegg til de andre egenskapene emballasjen har.

Av alle produkttypene i klassehierarkiet produseres det for tiden bare ting av disse tre: liten plastflaske med pant, liten nedbrytbar plastflaske med pant og stor nedbrytbar pappflaske med pant.

1a Klassehierarki

Lever en tegning av klassehierarkiet for de tre produkttypene som er beskrevet over som en besvarelse på denne oppgaven. Inkluder også superklasser og eventuelle grensesnitt.

Hva er en klasse egentlig?

En **klasse** er en *modellering* av noe (en ting eller et begrep).

Konkret

En klasse inneholder

- en **representasjon** (dvs variabler)
- én eller flere **konstruktører**
- diverse **operasjoner** (dvs metoder)

Hvilke klasser trenger jeg?

Emballasjefabrikken *Renpakk* skal lage et nytt **datasystem** (i Java) for å administrere sine **produkter**, og du har fått i oppdrag å lage **deler** av dette **systemet**.

Alt Renpakk produserer kalles **Emballasje**, og det er fire hovedtyper: **Glassemballasje**, **Metallemballasje**, **Plastemballasje** og **Pappemballasje**.

Noe av emballasjen til Renpakk er det pant på, og noe emballasje er nedbrytbar. Dette er egenskaper som kommer i tillegg til de andre egenskapene emballasjen har.

Av alle produkttypene i klassehierarkiet produseres det for tiden bare ting av disse tre: **liten plastflaske med pant**, **liten nedbrytbar plastflaske med pant** og **stor nedbrytbar pappflaske med pant**.

1a Klassehierarki

Lever en tegning av klassehierarkiet for de tre produkttypene som er beskrevet over som en besvarelse på denne oppgaven. Inkluder også superklasser og eventuelle grensesnitt.

Hva er *ikke* klasser i vår modell?

- «emballasjefabrikk» (kanskje i en annen sammenheng)
- «datasystem»
- «produkt»
- «del»
- «system»

Hva er et interface?

Et **interface** er en *egenskap*. Noen klasser har denne egenskapen, andre har den ikke.

Konkret

Et interface inneholder

- navn på diverse **operasjoner** (dvs metoder) vi garanterer at en implementasjon inneholder.

Hvilke *egenskaper* finner vi?

Emballasjefabrikken *Renpakk* skal lage et nytt datasystem (i Java) for å administrere sine produkter, og du har fått i oppdrag å lage deler av dette systemet.

Alt Renpakk produserer kalles Emballasje, og det er fire hovedtyper: Glassemballasje, Metallemballasje, Plastemballasje og Pappemballasje.

Noe av emballasjen til Renpakk er det **pant** på, og noe emballasje er **nedbrytbar**. Dette er egenskaper som kommer i tillegg til de andre egenskapene emballasjen har.

Av alle produkttypene i klassehierarkiet produseres det for tiden bare ting av disse tre: **liten** plastflaske med pant, **liten** nedbrytbar plastflaske med pant og **stor** nedbrytbar pappflaske med pant.

1a Klassehierarki

Lever en tegning av klassehierarkiet for de tre produkttypene som er beskrevet over som en besvarelse på denne oppgaven. Inkluder også superklasser og eventuelle grensesnitt.

Hvilke interface-er trenger vi?

- **Nedbrytbar**
- **Pant**

(Vi *kunne* også definert interface for **liten** og **stor**, men vi ikke har noen metoder å gi dem.)

Hva er en subklasse?

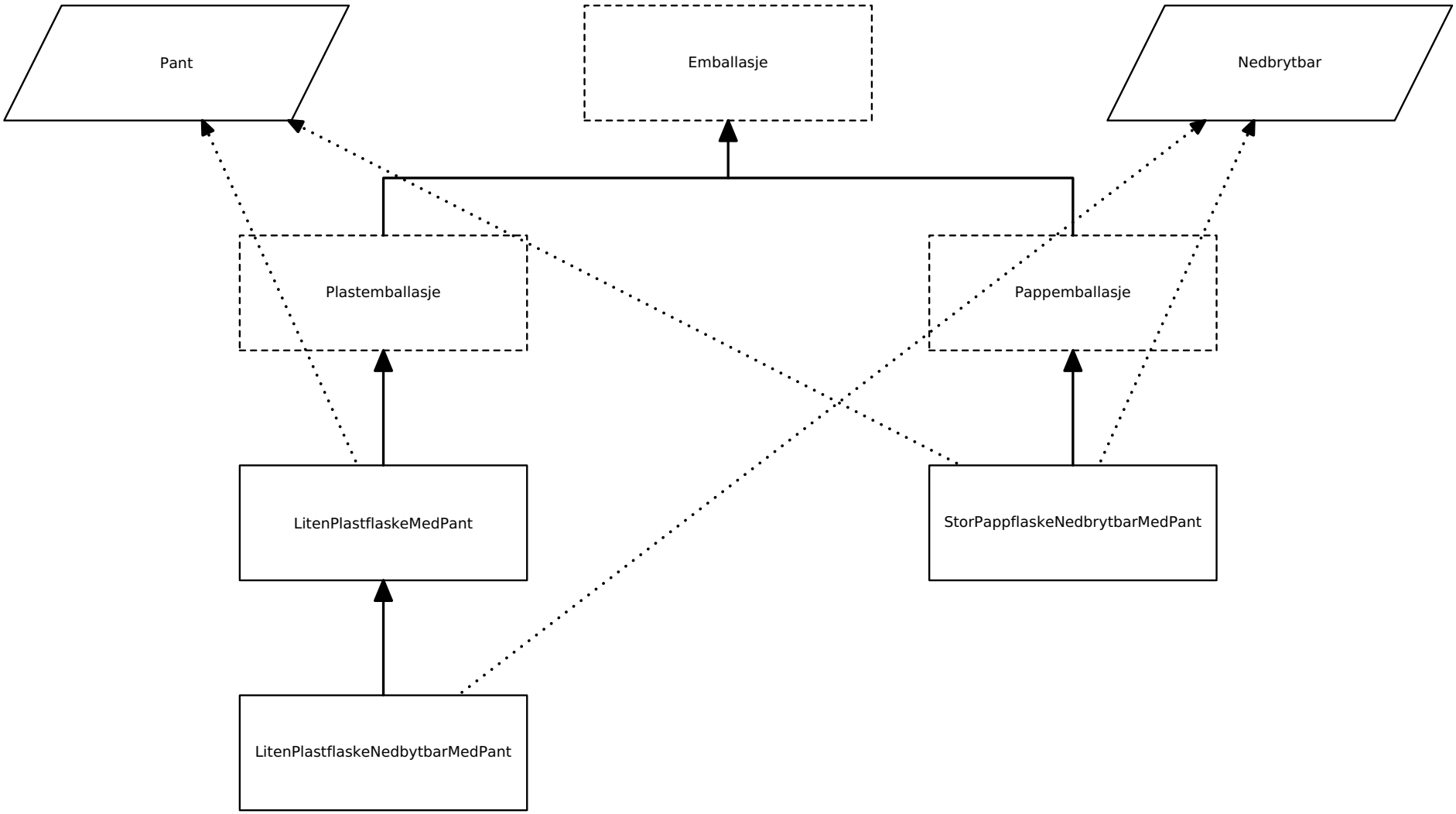
En **subklasse** er en *spesialisering* av en klasse, noe som skiller den av superklassen og eventuelt andre subklasser.

Konkret

En **subklasse** inneholder

- en utvidet representasjon (dvs flere variabler)
- flere konstruktører
- flere operasjoner (dvs metoder)
- redefinering av operasjoner (dvs @Override-de metoder)

Å sette alt sammen



Implementasjon

All emballasje har et **volum** (i kubikkcentimeter) og en tekst (String) som er en **produksjonsidentifikator**.

Plastemballasje har ingen flere egenskaper enn Emballasje mens Pappemballasje i tillegg har en **vekt** (i gram).

Når det er pant på en ting, må en kunne vite hvor stor **panten** er (i antall øre) og en kode (en tekst) som identifiserer **returordningen**.

Når noe er nedbrytbart, må en kunne vite hvor **lenge** (hvor mange år) det tar før tingen er gått i oppløsning.

Programmer de tre klassene nevnt i 1a og eventuelle superklasser og grensesnitt. Alle variable i alle klasser skal få verdier i det objektene opprettes. Det er derfor viktig at alle klasser har konstruktører med parametre der disse verdiene kan oppgis, unntatt panten på **små flasker** som alltid har samme verdi, en konstant med verdi **100 øre**. Når en konstruktør utføres, skal det skrives ut i terminalvinduet: «Konstruktoeren til klassen Xxx utfoeres», der Xxx er navnet på klassen.

Nå kan vi programmere

```
abstract class Emballasje {
    public double volum;
    public String produktId;

    public Emballasje(double vol, String id) {
        volum = vol;  produktId = id;
        System.out.println("Konstruktoeren til klassen Emballasje utfoeres.");
    }
}
```

```
abstract class Plastemballasje extends Emballasje {
    public Plastemballasje(double vol, String id) {
        super(vol, id);
        System.out.println("Konstruktoeren til klassen Plastemballasje utfoeres.");
    }
}
```

```
abstract class Pappemballasje extends Emballasje {
    public double vekt;

    public Pappemballasje(double vol, String id, double vekt) {
        super(vol, id);
        this.vekt = vekt;
        System.out.println("Konstruktoeren til klassen Pappemballasje utfoeres.");
    }
}
```

Nå kan vi programmere

```
interface Pant {
    int finnPant();
    String finnReturordning();
}
```

```
interface Nedbrytbar {
    double finnNedbrytningstid();
}
```

Nå kan vi programmere

```
class LitenPlastflaskeMedPant extends Plastemballasje implements Pant {
    public String returordning;

    public LitenPlastflaskeMedPant(double vol, String id, String retur) {
        super(vol, id);
        returordning = retur;
        System.out.println("Konstruktoeren til klassen " +
                           "LitenPlastflaskeMedPant utfoeres.");
    }

    public int finnPant() {
        return 100;
    }

    public String finnReturordning() {
        return returordning;
    }
}
```

Nå kan vi programmere

```
class LitenPlastflaskeNedbrytbarMedPant extends LitenPlastflaskeMedPant
        implements Nedbrytbar
{
    private double nedbrytningstid;

    public LitenPlastflaskeNedbrytbarMedPant(double vol, String id,
                                             String retur, double nedbryt)
    {
        super(vol, id, retur);
        nedbrytningstid = nedbryt;
        System.out.println("Konstruktoeren til klassen " +
                           "LitenPlastflaskeNedbrytbarMedPant utfoeres.");
    }

    public double finnNedbrytningstid() {
        return nedbrytningstid;
    }
}
```

Nå kan vi programmere

```
class StorPappflaskeNedbrytbarMedPant extends Pappemballasje
        implements Nedbrytbar, Pant
{
    private int pant;
    private String returordning;
    private double nedbrytningstid;

    public StorPappflaskeNedbrytbarMedPant(double vol, String id, double vekt,
                                           int pant, String retur, double nedbryt)
    {
        super(vol, id, vekt);
        this.pant = pant;
        returordning = retur;
        nedbrytningstid = nedbryt;
        System.out.println("Konstruktoeren til klassen " +
                           "StorPappflaskeNedbrytbarMedPant utfoeres.");
    }

    public int finnPant() {
        return pant;
    }

    public String finnReturordning() {
        return returordning;
    }

    public double finnNedbrytningstid() {
        return nedbrytningstid;
    }
}
```

Implementasjon (mer)

Skriv også en klasse (en hovedklasse) kalt BrukPant med en main-metode som oppretter ett objekt av hver av de klassene det kan lages objekter av. Finn på noen passende verdier til alle variablene i alle objektene. Programmet ditt skal kunne kompileres og kjøres.

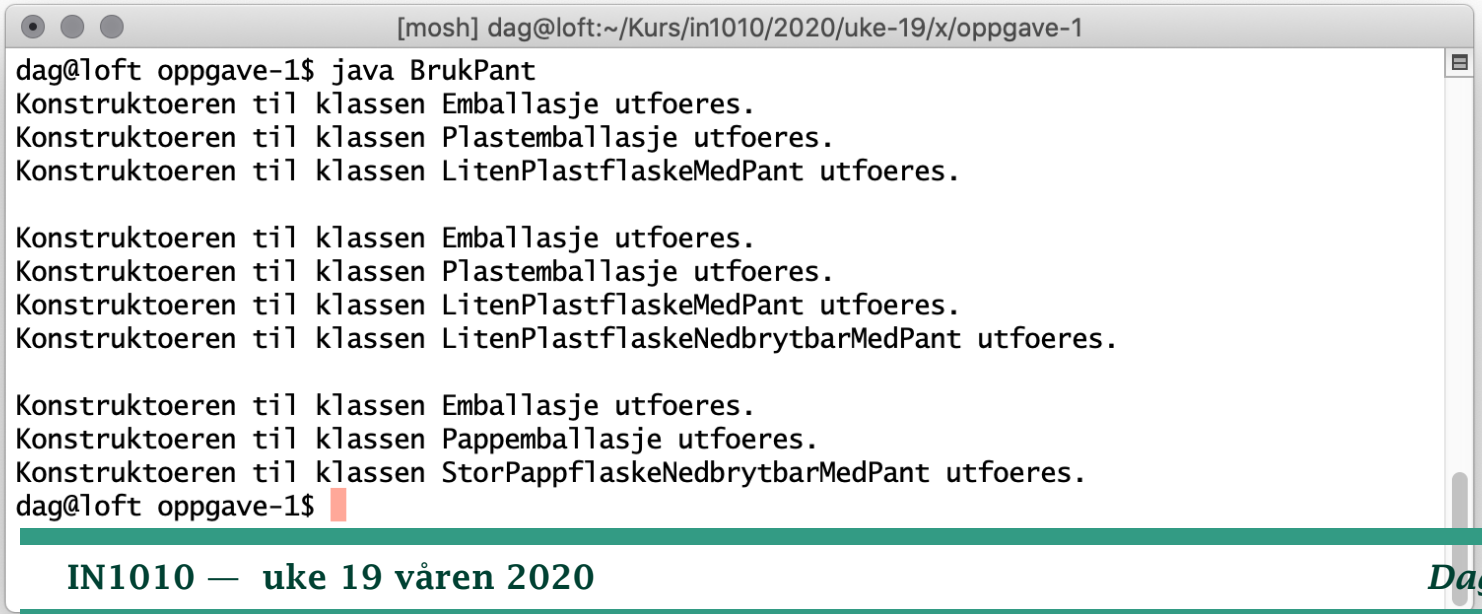
```
class BrukPant {
    public static void main(String[] arg) {
        Emballasje lqp = new LitenPlastflaskeMedPant(0.5, "LQP-h", "Plastretur");
        System.out.println();
        Emballasje lqnp = new LitenPlastflaskeNedbrytbarMedPant(0.5,
                                    "LQNP-h", "Plastretur", 27);
        System.out.println();
        Emballasje spnp = new StorPappflaskeNedbrytbarMedPant(3,
                                    "SPNP-3", 0.2, 5, "Papirkonteiner", 7);
    }
}
```

Implementasjon (til sist)

Kjør programmet du har laget og ta et bilde eller en skjermdump som viser kjøringen av dette programmet med alle linjene som blir skrevet ut i terminalvinduet.

NB!

Unngå å få med noe personinformasjon under eksamen!



Innsending

For å kunne levere svaret, men vi kombinere de tre filene til en ZIP-fil:

Linux zip oppg1.zip BrukPant.java fig.pdf skjermdump.png

MacOS Velg filer og control+klikk ⇒ Compress 3 items

Windows Velg filer og høyreklikk ⇒ New ⇒ Compressed (zipped) folder

(Se også https://www.uio.no/studier/eksamen/innlevering/levere-besvarelsen/zipfiler/hvordan-levere-zipfiler.html.)

Oppgave 2 Datastruktur

I denne oppgaven skal vi utvikle en klasse med en ny form for lagringsstruktur. Klassen skal hete Frekvens og skal kunne lagre **inntil 1000 tekster** (String-er). Konstruktøren skal ha én parameter: en array med tekstene som skal lagres. **Tekstene er alfabetisk sortert.**

Klassen Frekvens skal ha tre metoder:

1. void-metoden finnFlest skal finne hvilken tekst som forekommer flest ganger og lagre teksten og antallet forekomster i instansvariabler i Frekvens-objektet.

   (Hvis flere tekster forekommer like mange ganger, er det det samme hvilken som lagres.)
2. String-metode hentFlest henter teksten som ble lagret av et tidligere kall på finnFlest (se forrige punkt).
3. int-metoden hentAntall som henter antallet forekomster funnet i et tidligere kall på finnFlest (se punkt 1).

2a Fritt valgt datastruktur

Implementer klassene Frekvens og BrukFrekvens forklart over. I denne deloppgaven kan du bruke arrayer eller hvilke klasser du vil fra Javas bibliotek til å lagre tekstene.

Jeg har valgt å lagre data i en array.

```
class Frekvens {
    String[] data;
    String flest;
    int flestAntall;

    Frekvens(String[] initData) {
        data = new String[initData.length];
        for (int i = 0;  i < data.length;  i++)
            data[i] = initData[i];
    }

    String hentFlest() { return flest; }
    int hentAntall() { return flestAntall; }
```

Letingen etter den vanligste forekomsten blir helt rett frem når vi vet at array-en er alfabetisk sortert:

```
    void finnFlest() {
        String denne = data[0];
        int denneAntall = 1;

        flest = denne;  flestAntall = 1;
        for (int i = 1;  i < data.length;  i++) {
            if (denne.equals(data[i])) {
                denneAntall++;
            } else {
                denne = data[i];  denneAntall = 1;
            }
            if (denneAntall > flestAntall) {
                flest = denne;  flestAntall = denneAntall;
            }
        }
    }
}
```

Skriv også en klasse BrukFrekvens med et testprogram; det skal ha en main-metode som setter det hele i gang. Testprogrammet skal ha én parameter: navnet på en fil med tekster, én tekst per linje. **Denne filen vil være alfabetisk sortert**. Det skal opprette et Frekvens-objekt med innleste data og etterpå skrive ut hvilken tekst som forekommer flest ganger og hvor mange ganger det er.

I de fleste tilfeller blir innlesing i Java enklest ved å bruke Scanner.

```
class BrukFrekvens {
    public static void main(String[] arg) {
        String[] lager = new String[1000];
        int antILager = 0;

        if (arg.length != 1) {
            System.out.println("Usage: java BrukFrekvens datafil");
            System.exit(1);
        }
        try {
            Scanner s = new Scanner(new File(arg[0]));
            while (s.hasNextLine()) {
                lager[antILager++] = s.nextLine();
            }
        } catch (FileNotFoundException e) {
            System.out.println("Kan ikke lese " + arg[0] + "!");
            System.exit(2);
        }

        String[] lager2 = new String[antILager];
        for (int i = 0;  i < antILager;  i++) lager2[i] = lager[i];
        Frekvens tekster = new Frekvens(lager2);
        tekster.finnFlest();
        System.out.println("Det vanligste navnet er " + tekster.hentFlest() +
                           " (" + tekster.hentAntall() + " forekomster).");
    }
}
```

2b Énveis liste

I denne deloppgaven (og alle de etterfølgende deloppgavene) får du ikke lov å benytte ArrayList, HashMap eller andre lagringsstrukturer fra Java-biblioteket.

Du skal nå endre svaret ditt fra forrige deloppgave til å bruke en énveis linket liste til å lagre dataene.

En standard løsning med en lokal klasse Node.

```
class Frekvens {
    class Node {
        Node neste = null;
        String navn;

        Node(String n) { navn = n; }
    }

    Node forste = null;
    String flest;
    int flestAntall;

    Frekvens(String[] initData) {
        Node siste = null;

        forste = null;
        for (int i = 0;  i < initData.length;  i++) {
            if (forste == null) {
                forste = siste = new Node(initData[i]);
            } else {
                siste.neste = siste = new Node(initData[i]);
            }
        }
    }

    String hentFlest() { return flest; }
    int hentAntall() { return flestAntall; }
```

Letingen blir også helt standard:

```
    void finnFlest() {
        flest = null;
        flestAntall = 0;

        String denne = "";
        int denneAntall = 0;
        Node p = forste;
        while (p != null) {
            if (p.navn.equals(denne)) {
                denneAntall++;
            } else {
                denne = p.navn;  denneAntall = 1;
            }
            if (denneAntall > flestAntall) {
                flest = denne;  flestAntall = denneAntall;
            }
            p = p.neste;
        }
    }
}
```

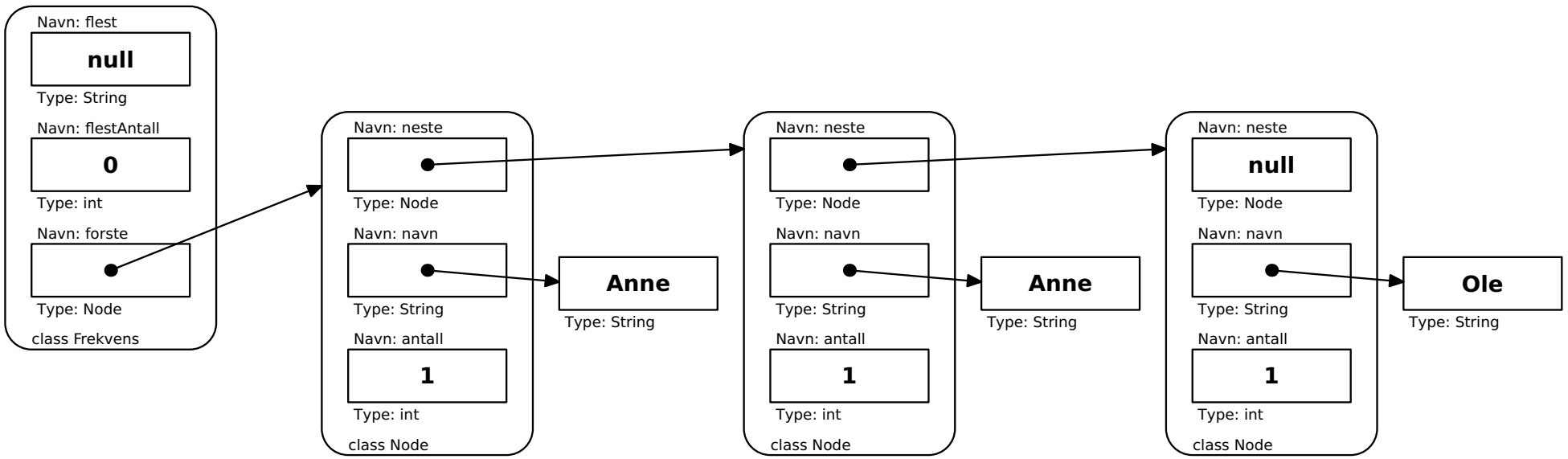
2c Énveis liste med antall

Du skal nå oppdatere koden fra forrige deloppgave til å lagre data i en **énveis liste** av objekter av en **indre klasse**. I disse objektene skal du ikke bare lagre String-en og nestepekeren, men også **antallet** ganger String-en forekommer. Ved initieringen av Frekvens er dette antallet **1** for alle objektene.

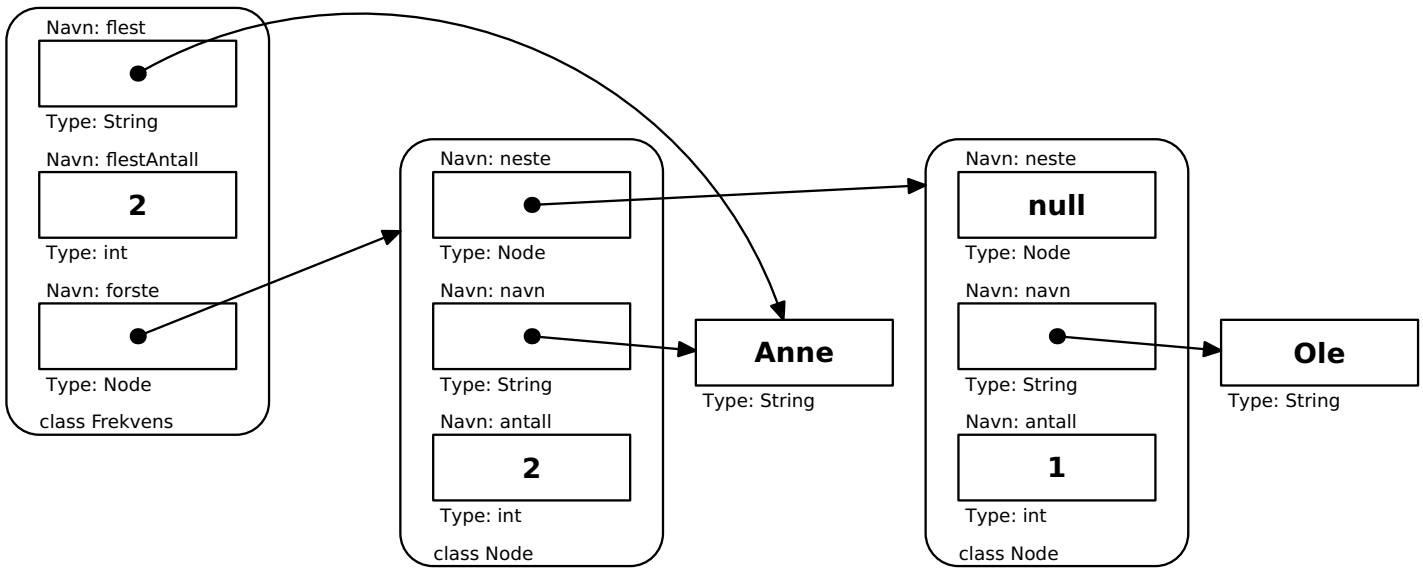
Du skal så utvide klassen Frekvens med en metode komprimer som forkorter listen ved å slå sammen etterfølgende like tekster. **(Husk at alle tekstene er alfabetisk sortert i datafilen.)**

Tegn et eksempel på datastrukturen **før** og **etter** en slik komprimering; velg nok objekter til at strukturen kommer klart frem. Bruk et tegneverktøy du liker, eller tegn på et ark og ta bilde av det.

Før komprimering og leting



Etter komprimering og leting



Den indre klassen Node blir som i forrige oppgave, bortsett fra at den nå også får et antall.
Konstruktøren for Frekvens blir identisk.

```
class Frekvens {
    class Node {
        Node neste = null;
        String navn;
        int antall = 1;

        Node(String n) { navn = n; }
    }

    Node forste = null;
    String flest;
    int flestAntall;
```

```
    Frekvens(String[] initData) {
        Node siste = null;

        forste = null;
        for (int i = 0;  i < initData.length;  i++) {
            if (forste == null) {
                forste = siste = new Node(initData[i]);
            } else {
                siste.neste = siste = new Node(initData[i]);
            }
        }
    }

    String hentFlest() { return flest; }
    int hentAntall() { return flestAntall; }
```

Metoden komprimer går gjennom listen og fjerner de Node-ene der etterfølgeren har samme tekst.

```
void komprimer() {
    Node p1 = forste, p2 = forste.neste;

    while (p2 != null) {
        if (p1.navn.equals(p2.navn)) {
            p1.antall++;  p1.neste = p2.neste;
        } else {
            p1 = p2;
        }
        p2 = p2.neste;
    }
}
```

Metoden finnFlest blir nå enklere:

```
void finnFlest() {
    flest = null;
    flestAntall = 0;

    Node p = forste;
    while (p != null) {
        if (p.antall > flestAntall) {
            flest = p.navn;  flestAntall = p.antall;
        }
        p = p.neste;
    }
}
```

2d Parallell leting

Du skal nå gå tilbake til deloppgave 2a. Ta utgangspunkt i koden du laget til den og la data i Frekvens være lagret i en array. (Du skal altså ikke bruke noen énveis liste.)

For å prøve å forbedre hastigheten skal du bruke tråder til å parallellisere tellingen av tekster som gjøres av finnFlest. Metoden skal opprette én eller flere tråder og la hver tråd lete i sin del av arrayen.

Oppgaver med tråder

I alle oppgaver med tråder må man først få oversikt over hva som skal skje:

1. Hovedtråden skal starte n tråder med angivelse av hvilken del av arrayer de skal lete i. (Da vil de ikke forstyrre hverandre.)
2. Når en tråd har funnet sin mest brukte tekst, skal den sjekke mot det globale resultatet og – om aktuelt – oppdatere det.
3. Når alle trådene er ferdige, skal hovedtråden skrive ut resultatet.

Til dette trenger vi følgende mekanismer:

- én ReentrantLock for å sikre det globale resultatet.
- én CountDownLatch for å kunne vente på de n trådene.

Letingen skjer altså i flere tråder.

```
class Leting implements Runnable {
    int nr, fra, til;

    Leting(int nr, int fra, int til) {
        this.nr = nr;  this.fra = fra;  this.til = til;
    }

    public void run() {
        System.out.println("Leteprosess nr " + nr + " leter i intervallet " +
                           fra + "-" + til + "...");

        String denne = data[fra];
        int denneAntall = 1;
        String max = denne;
        int maxAntall = denneAntall;
        for (int i = fra+1;  i <= til;  i++) {
            if (data[i-1].equals(data[i])) {
                denneAntall++;
            } else {
                if (denneAntall > maxAntall) {
                    max = denne;  maxAntall = denneAntall;
                }
                denne = data[i];  denneAntall = 1;
            }
        }
        if (denneAntall > maxAntall) {
            max = denne;  maxAntall = denneAntall;
        }
```

Når letingen i en tråd er ferdig, må resultatet sjekkes mot det globale som kanskje må oppdateres. Siden variablene som inneholder det globale resultatet er felles for trådene, blir aksessen en **kritisk region** som må beskyttes av låser.

```
        try {
            laas.lock();
            if (maxAntall > flestAntall) {
                flest = max;  flestAntall = maxAntall;
            }
        } finally {
            laas.unlock();
        }

        System.out.println("Leteprosess nr " + nr + " er ferdig.");
        barriere.countDown();
    }
```

Metoden finnFlest i hovedtråden deler opp arrayen og starter letetrådene.

> ***NB!*** *Når du deler opp arrayen, kan du ikke alltid dele den i like store deler. Du må kanskje justere grensene noe slik at sekvenser av samme tekst ikke havner i to forskjellige deler.*

```
void finnFlest() {
    final int ant_per_prosess = (data.length-1)/ant_prosesser + 1;
    barriere = new CountDownLatch(ant_prosesser);

    try {
        int pNr = 0;
        int fra = 0;
        while (fra < data.length) {
            pNr++;
            int til = Math.min(fra+ant_per_prosess-1, data.length-1);
            while (til<data.length-1 && data[til].equals(data[til+1]))
                til++;
            new Thread(new Leting(pNr, fra, til)).start();
            fra = til+1;
        }
```

Det kan hende at fordelingen av navn er så skjev at ikke alle letetrådene får nmoe å gjøre.

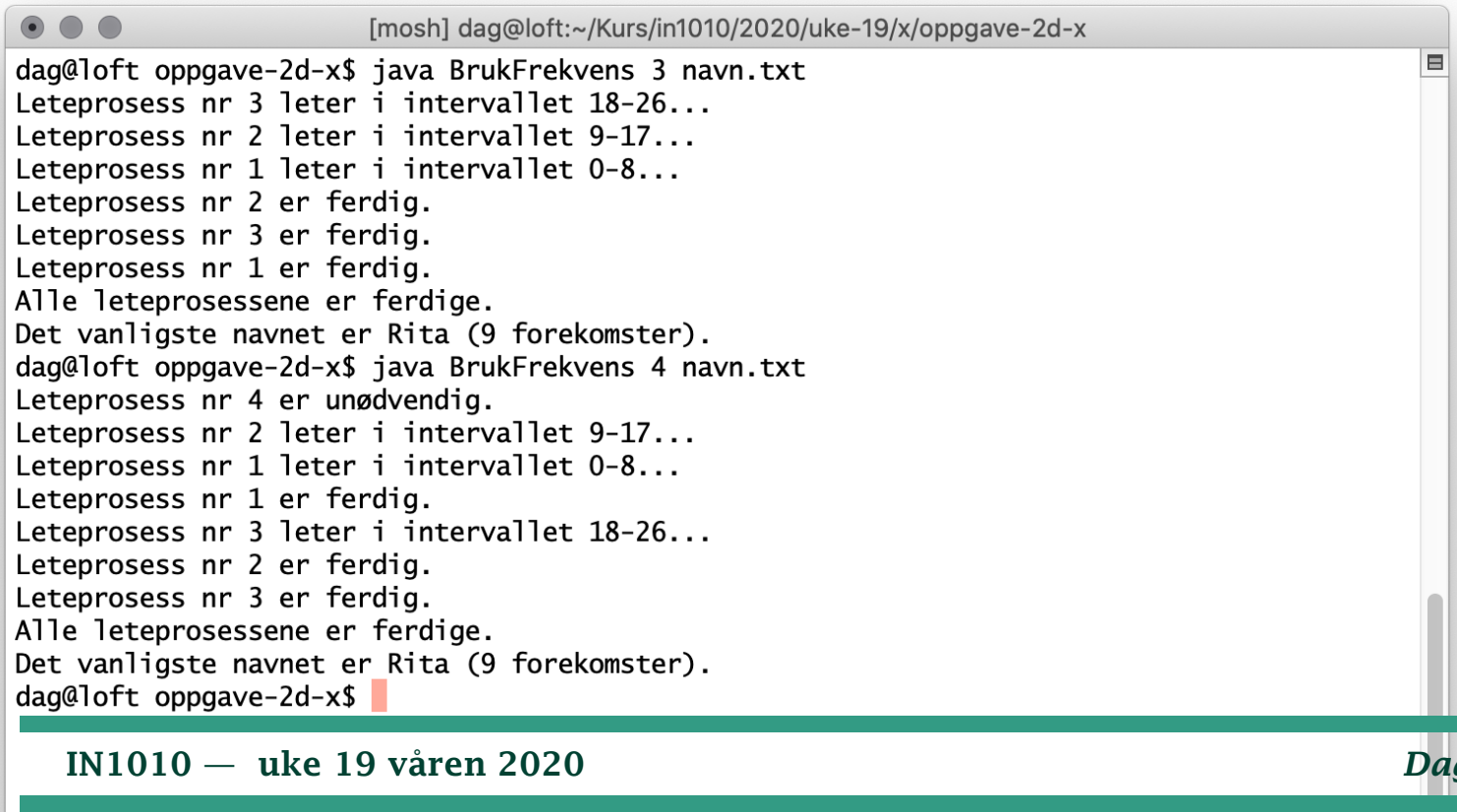
```
while (pNr < ant_prosesser) {
    pNr++;
    System.out.println("Leteprosess nr " + pNr + " er unødvendig.");
    barriere.countDown();
}
```

Til sist er det bare å vente . . .

```
barriere.await();
```

Dette programmet skal ha to parametre: antallet tråder og datafilen. Kjør derfor programmet to ganger med disse kommandoene

java BrukFrekvens 3 navn.txt
java BrukFrekvens 4 navn.txt



2e Generelle data

Du skal nå ta utgangspunkt i koden din fra del 2c (dvs den med metoden komprimer). Gjør klassen Frekvens *generisk* slik at den kan brukes til å lagre ulike typer data, ikke bare String-er.

```
class Frekvens<T extends Comparable<T>> {
    class Node {
        Node neste = null;
        T data;
        int antall = 1;
```

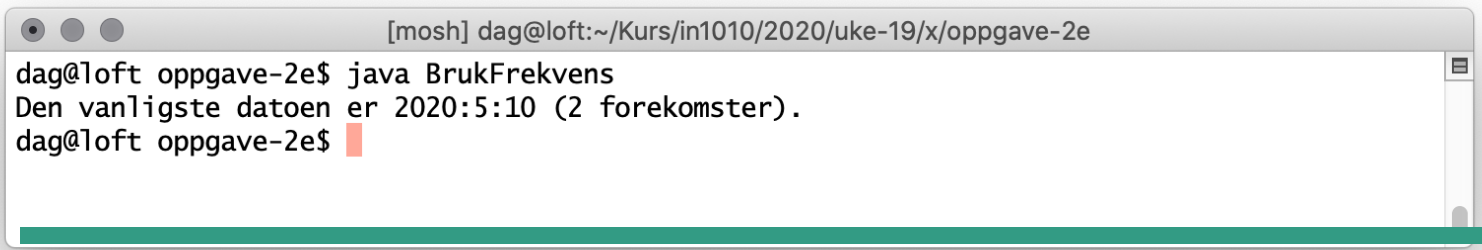
Opprett også en egen klasse du kan bruke som data.

```
class Dato implements Comparable<Dato> {
    int aa, m, d;

    Dato(int aa, int m, int d) {
        this.aa = aa;  this.m = m;  this.d = d;
    }

    @Override
    public int compareTo(Dato x) {
        if (aa != x.aa) return aa - x.aa;
        if (m != x.m) return m - x.m;
        return d - x.d;
    }

    @Override
    public String toString() {
        return aa + ":" + m + ":" + d;
    }
}
```

Deloppgave 2e

Du skal ikke lese data fra en fil denne gangen, men lage objekter av denne dataklassen og legge disse inn i Frekvens selv. Dette skal gjøres i main-metoden. Lag minst fem dataobjekter.

```
class BrukFrekvens {
    public static void main(String[] arg) {
        Frekvens<Dato> mineData = new Frekvens<Dato>();
        mineData.add(new Dato(2000,  1,  1));
        mineData.add(new Dato(2020,  5, 10));
        mineData.add(new Dato(2020,  5, 10));
        mineData.add(new Dato(2020,  5, 17));
        mineData.add(new Dato(2020,  7, 17));

        mineData.komprimer();
        mineData.finnFlest();
        System.out.println("Den vanligste datoen er " +
            mineData.hentFlest() + " (" +
            mineData.hentAntall() + " forekomster).");
    }
}
```



2f GUI-grensesnitt

Ta igjen utgangspunkt i koden fra 2c og gi programmet et GUI-grensesnitt.

Her trenger vi disse GUI-elementene:

- standardelementene Stage, Scene og Pane
- en Button til «Velg fil»
- en Text til «Vanligst:»
- en Text til «Antall:»
- en Button til «Stopp»

GUI-grensesnitt

```
public class BrukFrekvens extends Application {
    Stage mittTeater;
    Text resultat, resAntall;

    @Override
    public void start(Stage teater) {
        mittTeater = teater;

        Button velgFil = new Button("Velg fil");
        velgFil.setLayoutX(10);  velgFil.setLayoutY(15);
        velgFil.setOnAction(new FilBehandler());

        resultat = new Text("Vanligst:        ");
        resultat.setX(10);  resultat.setY(65);

        resAntall = new Text("Antall:         ");
        resAntall.setX(10);  resAntall.setY(85);

        Button stopp = new Button("Stopp");
        stopp.setLayoutX(10);  stopp.setLayoutY(110);
        stopp.setOnAction(new StoppBehandler());

        Pane kulisser = new Pane();
        kulisser.setPrefSize(120,150);
        kulisser.getChildren().add(velgFil);
        kulisser.getChildren().add(resultat);
        kulisser.getChildren().add(resAntall);
        kulisser.getChildren().add(stopp);

        Scene scene = new Scene(kulisser);

        teater.setTitle("Frekvens");
        teater.setScene(scene);
        teater.show();
}
```

Klassen FilBehandler gjør hele jobben:

```
class FilBehandler implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent e) {
        File f = new FileChooser().showOpenDialog(mittTeater);
        String[] lager = new String[1000];
        int antILager = 0;
        try {
            Scanner s = new Scanner(f);
            while (s.hasNextLine()) {
                lager[antILager++] = s.nextLine();
            }
        } catch (FileNotFoundException ex) {}

        String[] lager2 = new String[antILager];
        for (int i = 0;  i < antILager;  i++) lager2[i] = lager[i];
        Frekvens tekster = new Frekvens(lager2);
        tekster.komprimer();
        tekster.finnFlest();
        resultat.setText("Vanligst: " + tekster.hentFlest());
        resAntall.setText("Antall:    " + tekster.hentAntall());
    }
}
```

2g Frekvensoversikt

Til sist skal du ta utgangspunkt i koden du laget i 2c og utvide den til også å vise en oversikt over hvor mange forekomster det er av unike navn, 2 like navn, 3 like navn osv.

Oppgaven løses greit ved å lage en frekvenstabell:

```
void skrivOversikt() {
    int antForekomster[] = new int[flestAntall+1];
    Node p = forste;
    while (p != null) {
        antForekomster[p.antall]++;
        p = p.neste;
    }

    for (int i = flestAntall;  i > 0;  i--) {
        if (antForekomster[i] > 0)
            System.out.printf("%3d like navn: %d forekomster\n",
                              i, antForekomster[i]);
    }
}
```