

## Løsningsforslag eksamen

Merknad: I løsningene står det av og til små tall i margen, f.eks.:

**1** `Consumer<String> cs = a -> System.out.println(a);`

Ikke bry dere om disse tallen. Disse er kun ment for sensor, og har med poengregistrering å gjøre. De må ikke tolkes som vekting.

**Emnekode: DAT108**

**Emnenavn: Programmering og webapplikasjoner**

**Klasse: 2. klasse DATA / INF**

**Dato: 16. desember 2022**

---

Eksamensform: Skriftlig skoleeksamen (Wiseflow | FLOWlock)

Eksamenstid: 4 timer (0900-1300)

Antall eksamensoppgaver: 5

Antall sider (medregnet denne): 10

Antall vedlegg: Ingen

Tillatte hjelpemiddel: Ingen

Lærere:     Lars-Petter Helland (928 28 046) lph@hvl.no  
              Bjarte Kileng (909 97 348) bki@hvl.no

## Oppgave 1 (20% ~ 48 minutter) – Lambda-uttrykk og strømmer

For å få full score må løsningene ikke være unødig kompliserte, og det bør brukes metodereferanser der det er mulig.

- a) Nedenfor ser du 5 ulike  $\lambda$ -uttrykk. Skriv en setning for hver av disse der de tilordnes en variabel. Det vi er ute etter er datatypen til variablene, f.eks. om det er en `Consumer<String>`, en `Function<Integer, String>`, etc. ..

```
a -> System.out.println(a)    // der a er en String
(a,b) -> a.compareTo(b)       // der a og b er String-er
a -> a * a                    // der a er et heltall
a -> a > 0                    // der a er et heltall
(a,b) -> a + b                // der a og b er heltall
```

Løsningsforslag:

```
1 Consumer<String>          cs = a -> System.out.println(a);
1 Comparator<String>       cp = (a,b) -> a.compareTo(b);
                             (evt. BiFunction<String, String, Integer>)
1 Function<Integer, Integer> fu = a -> a * a;
                             (evt. UnaryOperator<Integer>)
1 Predicate<Integer>       pr = a -> a > 0;
                             (evt. Function<Integer, Boolean>)
1 BiFunction<Integer, Integer, Integer> bo = (a,b) -> a + b;
                             (evt. BinaryOperator<Integer>)
```

b) Anta at du har metoden **utplukk**, metoden **main** og resultatet **[3, 6, 9]** av å kjøre main:

```
List<Integer> utplukk(List<Integer> liste, ??? b) {  
    return liste.stream().filter(b).toList();  
}  
  
void main(...) {  
    List<Integer> liste = List.of(1,2,3,4,5,6,7,8,9);  
    List<Integer> resultat = utplukk(liste, ???);  
    System.out.println(resultat);  
}  
  
[3, 6, 9]
```

Hva skal stå der det står ??? i parameterlisten (dvs. typen til den formelle parameteren b), og hva skal stå der det står ??? i metodekallet (dvs. den aktuelle parameteren)?

Løsningsforslag:

- 2**      **Predicate<Integer>** (typen til formell parameter, brukt i filter(b))
- 3**      **x -> x % 3 == 0**      (aktuell parameter, om et tall er delelig med 3)

I oppgave c) og d) skal du jobbe med en liste av biler, f.eks.:

```
List<Bil> biler = List.of(
    new Bil("EK 12345", "Tesla model Y"),
    new Bil("EV 52345", "Tesla model Y"),
    ...
    new Bil("SV 12346", "Mazda 5"),
    new Bil("SU 24680", "Volvo 240"),
    new Bil("EL 24683", "Nissan Leaf"));
```

- c) Bruk streams til å lage en ny liste av alle elbilene (de som har **skilt** som starter med 'E'). Legg svaret inn i en variabel.

Løsningsforslag:

```
5 List<Bil> elbiler = biler.stream()
    .filter(b -> b.getSkilt().startsWith("E")).toList();

    (evt. ...collect(Collectors.toList()) pre Java 16)
```

- d) Bruk streams til å skrive ut på skjermen (uten duplikater) alle **modell**navn på elbilene, én linje per bilmodell. Bruk gjerne svaret fra forrige spørsmål i løsningen.

Løsningsforslag:

```
5 elbiler.stream()
    .map(Bil::getModell).distinct().forEach(System.out::println);

    (evt. ...map(b -> b.getModell()) )
```

I oppgave e), f) og g) skal du jobbe beregning av strømregning for en liste med kunder. For enkelhets skyld kan vi si at data om hver kunde kun er **navn** og **forbruk** (i kWh), men vi får tenke oss at det også kunne har vært andre data her. En liste kan f.eks. se slik ut:

```
List<Kundedata> kundeliste = List.of(
    new Kundedata("Arne", 1234),
    new Kundedata("Per", 2234),
    new Kundedata("Pål", 1000),
    new Kundedata("Emma", 4000),
    new Kundedata("Ine", 5234),
    new Kundedata("Tone", 1111));
```

- e) Bruk streams til å beregne totalforbruket (i kWh) for alle kundene i kundelisten og legg svaret inn i en variabel.

Løsningsforslag:

Rent logisk er løsningen slik:

```
int totforbruk = kundeliste.stream().map(Kundedata::getForbruk).sum();
```

Siden sum() kun kan anvendes på tall må vi teknisk bruke mapToInt():

```
3 int totforbruk = kundeliste.stream().mapToInt(Kundedata::getForbruk).sum();
    (evt. ...mapToInt(k -> k.getForbruk())... )
```

Alternativt kan vi bruke map() og reduce():

```
int totalforbruk = kundeliste.stream().map(Kundedata::getForbruk)
    .reduce(0, (tot, f) -> tot + f);
    (evt. uten map: ...reduce(0, (tot, k) -> tot + k.getForbruk()). )
```

Strømselskapet ønsker å se på ulike modeller for prising, og hvordan dette slår ut på selskapets inntekter.

- f) Lag en metode `beregnTotalInntekt(...)` som tar inn en kundeliste og en funksjon for prisberegning for en enkelt kunde, og som regner ut total inntekt med dette som input.

Merknader til oppgave:

Noen hadde misforstått oppgaveteksten, som var ment å beregne total inntekt for strømselskapet basert på kundelisten, og ikke strøminntekt for én enkelt kunde.

Ellers var det også noen som tenkte at beregning av pris for en enkelt kunde bare er avhengig av forbruk/kWh, og ikke andre kundedata. Da kan beregningsfunksjonen få en litt annen/enklere signatur. Her var eksemplet nok forenklet litt for mye. Tanken var at det i teorien også kunne være andre data involvert, slik som avtaletype, etc...

Løsningsforslag:

Rent logisk er løsningen slik:

```
public double beregnTotalInntekt(
    List<Kundedata> kundeliste, Function<Kundedata, Double> prisKalku) {
    return kundeliste.stream().map(prisKalku).sum();
}
```

Men tilsv. som i e) må vi bruke `mapToDouble()` for å kunne bruke `sum()`. Og som om kke det er nok, så forventer `mapToDouble()` en `ToDoubleFunction`, ikke en vanlig `Function`.

En teknisk fungerende løsning kan da f.eks. være slik:

```
public double beregnTotalInntekt(
    List<Kundedata> kundeliste, ToDoubleFunction<Kundedata> prisKalku) {
    return kundeliste.stream().mapToDouble(prisKalku).sum();
}
```

En vanlig `Function` kan også brukes, men da med `apply()`, slik:

```
4 public double beregnTotalInntekt(
    List<Kundedata> kundeliste, Function<Kundedata, Double> prisKalku) {
    return kundeliste.stream().mapToDouble(k -> prisKalku.apply(k)).sum();
}
```

Evt. kan vi erstatte `sum()` med `reduce()`:

```
public double beregnTotalInntekt(
    List<Kundedata> kundeliste, Function<Kundedata, Double> prisKalku) {
    return kundeliste.stream().map(prisKalku).reduce(0.0, (t, b) -> t + b);
}
```

Alternativt kan også en tradisjonell imperativ algoritme brukes:

```
public double beregnTotalInntekt(  
    List<Kundedata> kundeliste, Function<Kundedata, Double> prisKalku) {  
  
    double total = 0.0;  
    for (Kundedata k : kundeliste) {  
        total += prisKalku.apply(k);  
    }  
    return total;  
}
```

Alle nevnte løsninger gir full score.

- g) Vis hvordan beregnTotalInntekt(...) kan brukes med et eksempel der prisberegningen er en flat sats på 1.50 kr per kWh. (Altså slik at f.eks. Pål får en regning på 1500,-)

Merknader til oppgave:

Eksemplet her med Pål og 1500,- var nok også en kilde til misforståelse. Det var fremdeles tenkt at totalinntekten skulle beregnes for kundelisten.

Eksemplet var ment som en "hjelp" til å forstå prisberegningen.

Løsningsforslag:

```
3 double total = beregnetTotalInntekt(kundeliste, k -> 1.5 * k.getForbruk());
```

## Oppgave 2 (20% ~ 48 minutter) – JavaScript

- a) JavaScript i nettleser har bla. objektene **window**, **navigator**, **history** og **screen**. For hvert av disse objektene, hva kan vi bruke objektet til?

Du trenger ikke huske navn på egenskaper.

Løsningsforslag:

**window**: Objektet representerer nettleservinduet som viser websiden. Alle globale variabler, funksjoner, objekter og klasser tilhører **window**. Objektet lar oss f.eks. finne størrelse på nettleservinduet og har også metoder for å åpne og arbeide med vinduer, timer metoder, osv.

**navigator**: Objektet gir oss egenskapene til nettleseren som viser websiden, f.eks. nettlesertype, språk, operativsystem ol.

**history**: Objektet har bla. metoder for å navigere framover og bakover i historielisten til nettleser og navigere til nye adresser.

**screen**: Objektet har egenskaper, metoder og hendelser knyttet til datamaskinen sin skjerm. Vi kan finne skjermopløsning og fargedybde og få kjørt metoder når skjermen roteres.

- b) JavaScript og datatyper:

- i. JavaScript betegnes som et type-svakt språk (*weakly typed, loosely typed*).

Hva betyr dette i praksis i JavaScript? Når bestemmes en variable sin type i JavaScript?

Løsningsforslag:

Type til variabel er ikke konstant, men kan endres under kjøring.

Type til variabel blir bestemt ved tilordning av data. Observer at variabel vil til enhver tid ha en type.

- ii. JavaScript sine 7 primitive datatyper inkluderer **string**, **number**, **boolean** og **symbol**.

Når brukes datatypen **symbol**? Demonstrer bruken av **symbol** med et kodeeksempel.

Løsningsforslag:

Datatypen brukes for å identifisere objekter, og har litt samme oppgave som primærnøkkel i databasesystem. Symbol-verdien er i seg selv er uinteressant, og kan ikke skrives ut.

Eksempelet under viser bruk av funksjonen *Symbol* for å opprette en ny verdi av type **symbol**. Hver bruk av *Symbol* funksjonen lager en ny unik verdi.

```
const student = {  
  id : Symbol () ,  
  navn : " Ole Olsen "  
};
```



- c) En JavaScript klasse **Studentarkiv** lar foreleser holde oversikt over sine studenter. En student er registrert med *id*, *etternavn*, *fornavn* og eventuelt *tlf* som er en liste med 0 eller flere telefonnumre.

Klassen **Studentarkiv** har følgende offentlige (public) metoder:

- *nystudent(student)*: Metoden legger inn en ny student i arkivet.

Parametre:

- Inn-parameter: **object**
- Returverdi: **boolean** eller verdien *null*

Inn-parameter *student* er et objekt med egenskaper *id*, *etternavn*, *fornavn* og eventuelt også *tlf*:

- *id* er et heltall av type **number** som er unikt for hver student,
- *etternavn* og *fornavn* er av type **string**,
- *tlf* er en forekomst av **Array** med telefonnumre, der hvert telefonnummer er en **string**.

Dersom formatet på inn-parameter *student* er feil eller en student med gitt id allerede finnes blir ikke arkivet oppdatert.

- Returverdi er *null* hvis formatet på inn-parameter *student* er feil.
- Hvis student med gitt id allerede finnes i arkivet er returverdien *false*.
- Hvis arkivet blir oppdatert med en ny student returneres verdien *true*.

- *harteleson(id, telefonnummer)*: Metoden sjekker om student er registrert med et gitt telefonnummer.

Parametre:

- Inn-parametre: **number, string**
- Returverdi: **boolean** eller verdien *null*

Metoden returnerer verdien *null* hvis det ikke finnes noen student med den angitte id. Returverdi er *true* hvis studenten er registrert med det gitte telefonnummeret, ellers *false*.

- *nytelefon(id, nummer)*: Metoden legger til et nytt telefonnummer for student.

Parametre:

- Inn-parametre: **number, string**
- Returverdi: **boolean** eller verdien *null*

Metoden returnerer verdien *null* hvis det ikke finnes noen student med den angitte id. Returverdi er *false* hvis studenten allerede er registrert med det gitte telefonnummeret. Returverdi er *true* hvis det nye telefonnummeret ble lagt til for studenten.

- *eksporterdata()*: Metoden returnerer en tekst med alt innhold i arkivet. Semikolon skiller feltene, og linjeslutt skiller studentene.

Kodeeksempelet nedenfor demonstrerer bruk av **Studentarkiv**:

```
const arkiv = new Studentarkiv();
const ole = arkiv.nystudent({
  id: 101,
  etternavn: "Olsen",
  fornavn: "Ole",
  tlf: ["112 23 344", "323 22 323"]
});
const anne = arkiv.nystudent(
  { id:106,etternavn:"Annesen",fornavn: "Anne" }
);
const oletelefon = arkiv.hartelefon(101, "112 23 344");
const annetelefon = arkiv.hartelefon(106, "767 44 333");
const arkivdata = arkiv.eksporterdata();
```

Etter at koden over har kjørt har konstantene følgende verdier:

- Konstantene *ole*, *anne* og *oletelefon* er *true*.
- Konstanten *annetelefon* er *false*,
- Konstanten *arkivdata* inneholder følgende tekst:

```
101;Olsen;Ole;112 23 344;323 22 323
106;Annesen;Anne
```

**Oppgave:** Skriv JavaScript-koden for **Studentarkiv** i samsvar med teksten over.

**Hjelp:**

- Både **Map**, **Set** og **Array** har en metode *forEach*.
- **Map** og **Set** sin metode *values* returnerer en iterator over alle verdier.
- **Map** sin metode *keys* returnerer en iterator over alle nøkler.
- **Array** sin statiske metode *from*, og også spre (*spread*) operator kan kopiere et itererbart objekt til en **Array**.
- **Map** har bla. metoder *has*, *set*, *get* og *delete*.
- **Set** har bla. metoder *has*, *add* og *delete*.
- Operator *typeof* returnerer en **string** som angir operanden sin type.
- Operator *instanceof* tester om operand er en forekomst av en klasse, også via arv.

Løsningsforslag:

```
class Studentarkiv {

  #arkiv = new Map();

  nystudent(student) {
    const { id, etternavn, fornavn, tlf: liste } = student;
    if (typeof id !== "number") return null;
    if (this.#arkiv.has(id)) return false;
    if (liste === undefined) {
      this.#arkiv.set(id, { etternavn, fornavn });
    } else {
      const tlf = new Set();
      liste.forEach(nummer => { tlf.add(nummer) });
      this.#arkiv.set(id, { etternavn, fornavn, tlf });
    }
    return true;
  }

  hartelefon(id, telefonnummer) {
    if (!this.#arkiv.has(id)) return null;
    const tlf = this.#arkiv.get(id).tlf;
    if (tlf === undefined) return false;
    return (tlf.has(telefonnummer));
  }

  nytelefon(id, nummer) {
    if (!this.#arkiv.has(id)) return null;
    if (this.#arkiv.get(id).tlf === undefined) {
      this.#arkiv.get(id).tlf = new Set();
    }
    const tlf = this.#arkiv.get(id).tlf;
    if (tlf.has(nummer)) return false;
    tlf.add(nummer);
    return true;
  }

  eksporterdata() {
    /**
     * Kunne brukt iterator-metoder her, men det har vi ikke
     * brukt i kurset. Omformer derfor til forekomst av Array.
     */
    const keys = Array.from(this.#arkiv.keys());
    if (keys.length === 0) return "";
    let utskrift = this.#eksportlinje(keys[0]);
    for (let i = 1; i < keys.length; ++i) {
      utskrift += `\n${this.#eksportlinje(keys[i])}`;
    }
    return utskrift;
  }
}
```

```
#eksportlinje(id) {  
  const { etternavn, fornavn, tlf } = this.#arkiv.get(id);  
  let linje = `${id};${etternavn};${fornavn}`;  
  if (tlf !== undefined) {  
    tlf.forEach(  
      nummer => { linje += `;${nummer}` }  
    );  
  }  
  return linje;  
}  
}
```

### Oppgave 3 (10% ~ 24 minutter) – Tråder

Vi ønsker å lage et program som beregner store Fibonacci-tall, f.eks. `fib(50)`. Du trenger ikke å tenke på hvordan utregningen gjøres. Poenget er at utregningen kan ta litt tid ( $O(2^n)$ ) hvis vi bruker en enkel/naiv algoritme.

For å ha en følelse av at programmet gjør noe mens beregningen foregår skal det skrives ut en prikk hvert sekund så lenge utregningen foregår. Når beregningen er ferdig skal svaret skrives ut på skjermen og programmet avslutte.

Eksempel på beregning og utskrift av `fib(48)`, som tok ca. 30 sekunder:

```
Beregner fib(48): .....  
Svar: 4807526976
```

Vi kan anta at vi har en metode **`long fib(int n)`** som vi kan gjøre et metodekall til for å få utført selve beregningen. (Det er denne metoden som bruker lang tid.)

Du skal skrive `main()`, som skal gjøre følgende:

- Tallet vi skal bruke i kjøringen (f.eks. 50) kan være en hardkodet konstant.
- Det skal opprettes og startes en tråd som gjør beregningen og skriver ut svaret.
- Det skal skrives ut en prikk hvert sekund så lenge den andre tråden **`isAlive()`** (som er en metode i `Thread`-klassen).

For å få full score må løsningen ikke være unødvendig komplisert.

Løsningsforslag:

```
10 public static void main(...) throws InterruptedException {  
    final int N = 50;  
  
    Thread fibCalc = new Thread(() -> System.out.println("\nSvar: " + fib(N)));  
  
    System.out.print("Beregner fib(" + N + "): ");  
    fibCalc.start();  
  
    while (fibCalc.isAlive()) {  
        System.out.print(".");  
        Thread.sleep(1000);  
    }  
}
```

## Oppgave 4 (10% ~ 24 minutter) – Passord

Korrekt håndtering og lagring av brukers passord er helt nødvendig av sikkerhets- og personvern hensyn.

Beskriv i tekniske termer hvordan brukers valgte passord bør prosesseres og lagres (i database) ved f.eks. oppretting av ny brukerkonto på en tjeneste. Fortell også hvordan de ulike skrittene i prosessen er med på å beskytte brukerens passord mot cracking.

### Merknader til oppgave:

Det er en del momenter rundt passordhåndtering som ikke er med i løsningsforslaget, f.eks. å gi råd om eller kreve en viss passordstyrke, å kun overføre passord via en kryptert forbindelse (HTTPS), etc ...

Besvarelser som har trukket inn slike ting har fått et par plusspoeng for det, selv om hovedpoenget var tenkt å være hvordan man lagrer passord for å forhindre tilgang og cracking.

### Løsningsforslag:

#### Prosess:

- 2 - Salte (bruk individuelt salt av en viss størrelse som legges til passord), deretter
- 2 - Hashe (bruk sikker og treg hash-algoritme til å hashe passord+salt)
- 2 - Salt og hash lagres i databasen.

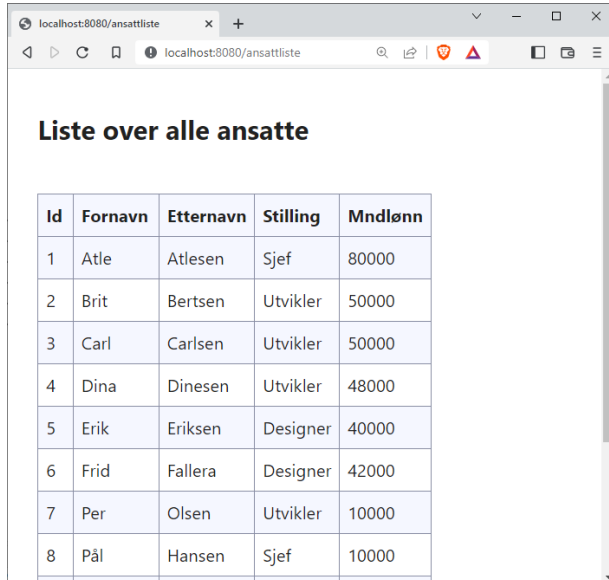
#### Beskyttelse mot cracking:

- 2 - Hashing generelt er en enveis funksjon, og lagring av hash betyr at passordet ikke kan "dekrypteres" eller utledes. En cracker MÅ prøve seg frem. En treg hash-algoritme vil gjøre crackingen mer tidkrevende.
- 2 - Salting hjelper mot muligheten til å forhåndsberegne milliarder av hasher som matcher mulige passord opp til en viss lengde. Passord pluss salt vil komme i så mange kombinasjoner at det er praktisk umulig å beregne hash for dem. I tillegg vil individuelle salt gjøre at like passord får ulik hash.

## Oppgave 5 (40% ~ 96 minutter) – Web backend med Spring MVC

Du skal lage deler av en applikasjon for å holde orden på ansatte i en bedrift. I hovedsak er det to ulike brukstilfeller vi skal se på:

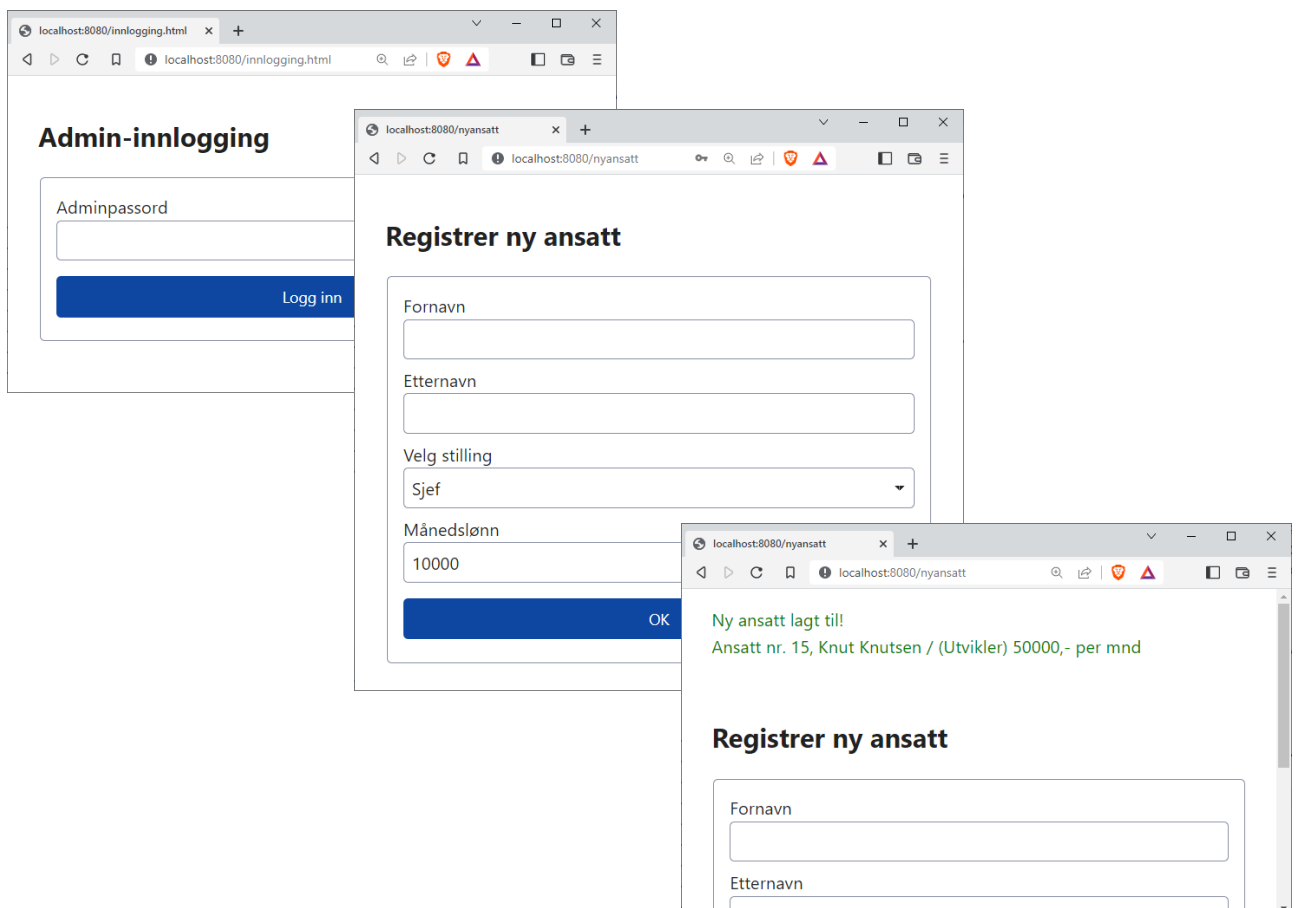
1. Få se en oversikt over alle ansatte



The screenshot shows a web browser window with the address bar at localhost:8080/ansattliste. The page title is "Liste over alle ansatte". Below the title is a table with 5 columns: Id, Fornavn, Etternavn, Stilling, and Mndlønn. The table contains 8 rows of employee data.

Id	Fornavn	Etternavn	Stilling	Mndlønn
1	Atle	Atlesen	Sjef	80000
2	Brit	Bertsen	Utvikler	50000
3	Carl	Carlsen	Utvikler	50000
4	Dina	Dinesen	Utvikler	48000
5	Erik	Eriksen	Designer	40000
6	Frid	Fallera	Designer	42000
7	Per	Olsen	Utvikler	10000
8	Pål	Hansen	Sjef	10000

2. Registrere en ny ansatt. Denne funksjonen krever at man er 'innlogget som admin'.



The image shows three overlapping screenshots of a web application. The top-left screenshot shows the "Admin-innlogging" form with a text input for "Adminpassord" and a blue "Logg inn" button. The middle screenshot shows the "Registrer ny ansatt" form with inputs for "Fornavn", "Etternavn", a dropdown for "Velg stilling" (set to "Sjef"), and a text input for "Månedslønn" (set to "10000"), with an "OK" button. The bottom-right screenshot shows a confirmation message: "Ny ansatt lagt til! Ansatt nr. 15, Knut Knutsen / (Utvikler) 50000,- per mnd", followed by another "Registrer ny ansatt" form.

Litt mer detaljer om hvordan applikasjonen skal virke:

- Ansatte er lagret i databasen i en tabell kalt **ansatt** med attributter tilsv. det første skjermbildet.
- **Ansatt.id** genereres automatisk av databasen.
- Adminpassordet er oppgitt som **app.adminPassord** i **application.properties**
- Det er ingen direkte utlogging. I stedet blir admin logget ut automatisk etter et antall sekunder med inaktivitet, oppgitt i **app.adminTimeout** i **application.properties**
- Feil admin-passord gir innlogginsskjemaet på nytt uten noe mer info.
- Forsøk på å registrere en ansatt uten å være innlogget som admin gir innlogginsskjemaet på nytt uten noe mer info.
- Drop-down-listen der man velger stilling skal være generert dynamisk ut fra hvilke stillinger som finnes i databasen.
- Ugyldige data ved registrering av ny ansatt skal gi registreringsskjemaet tilbake med en (rød) melding (øverst) om at data ikke er gyldige. Dette er ikke vist på skjermbildene over. Du kan selv bestemme krav til gyldige data. (Det holder f.eks. med at String-data skal være av en viss minimumslengde, og at lønn må være positiv.)
- Vellykket registrering skal gi registreringsskjemaet tilbake med en (grønn) melding (øverst) om at ny ansatt er registrert, se skjermbilde.

Forslag til URL-er, Controllere og Views:

- **/ansattliste, /nyansatt og /innlogging**
- Én controller for henting av ansattlisten
- Én controller for henting av registreringsskjema og for registrering / lagring av nyansatt
- Én controller for admininnlogging
- **ansattliste.jsp, nyansattskjema.jsp og innlogging.html** (den siste kan være statisk)

Hjelpeklasser som kan brukes i controllerne:

- **@Service AnsattDbService** (som er knyttet til databasen) med metodene
  - List<Ansatt> finnAlleAnsatte()
  - List<String> finnStillinger()
  - Ansatt registrerNyAnsatt(String fornavn, String etternavn, String stilling, Integer lønn)
- **@Service LogginnService** med metodene
  - void loggInn(HttpServletRequest request)
  - erLoggetInn(HttpServletRequest request)



For å få full score må du løse oppgaven på best mulig måte i hht. det som er gjennomgått i kurset, dvs. **Spring MVC / Spring Boot, Model-View-Controller, Post-Redirect-Get, EL og JSTL** i JSP-ene, **robusthet, ufarliggjøring** av brukerinput, **god bruk av hjelpeklasser, elegant kode**, osv ...

Oppgaver:

- a) Skriv controlleren for henting av ansattlisten

Merknader til oppgave:

Det er litt uklart fra oppgavetekst om denne krever at man er innlogget. Begge tolkninger kan gi full score.

Ellers virket dette som en grei oppgave. Mange fikk full score.

... og glad er jeg for det siden Spring MVC var nytt pensum i år. :)

Løsningsforslag:

```
1 @Controller
1 @RequestMapping("/ansattliste")
  public class AnsattlisteController {

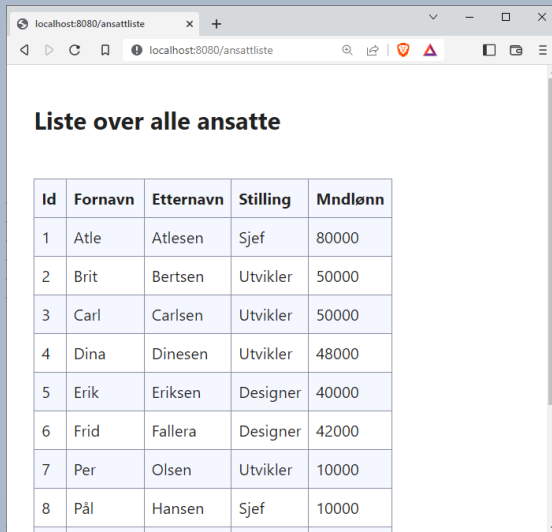
1     @Autowired
1     AnsattDbService ansattService;

1     @GetMapping
    public String visAnsattliste(Model model) {

2         List<Ansatt> ansatte = ansattService.finnAlleAnsatte();
2         model.addAttribute("ansattliste", ansatte);
1         return "ansattliste"; // spring.mvc.view.suffix=.jsp
    }
}
```

b) Skriv ansattliste.jsp

Merknader til oppgave:



Id	Fornavn	Etternavn	Stilling	Mndlønn
1	Atle	Atlesen	Sjef	80000
2	Brit	Bertsen	Utvikler	50000
3	Carl	Carlsen	Utvikler	50000
4	Dina	Dinesen	Utvikler	48000
5	Erik	Eriksen	Designer	40000
6	Frid	Fallera	Designer	42000
7	Per	Olsen	Utvikler	10000
8	Pål	Hansen	Sjef	10000

For å få god score må ting henge sammen!, dvs.:

```
# items i <c:forEach> må samsvare med attributt satt i controller, f.eks.:
model.addAttribute("ansattliste", ...);
...
... items="${ansattliste}" ...

# navn brukt inni løkke må samsvare med navn satt i <c:forEach>, f.eks.:
<c:forEach var="a" ...
...
... ${a.id} ...
```

Løsningsforslag:

```
1 <html><body>
  <h3>Liste over alle ansatte</h3>
  <table>
    <tr><th>Id</th><th>Fornavn</th><th>Etternavn</th>
    <th>Stilling</th><th>Mndlønn</th></tr>
3 + 3   <c:forEach var="a" items="${ansattliste}">
3       <tr><td>${a.id}</td>
        <td>${a.fornavn}</td>
        <td>${a.etternavn}</td>
        <td>${a.stilling}</td>
        <td>${a.mndlonn}</td></tr>
      </c:forEach>

  </table>
</body></html>
```

- c) Skriv controlleren med GET-mappingen som har ansvar for henting av registreringsskjema.  
(POST-mappingen i denne controlleren kommer i et senere delspørsmål.)

Merknader til oppgave:

Oppgaveformuleringen her var kanskje litt diffus, men tanken er at GET til denne controlleren skal (som vanlig) **klargjøre det som trengs for visning** og deretter "overlate ballen" til viewet. Hva trengs av klargjøring? Jo, **listen av stillinger** som man kan velge mellom ved registrering av ny ansatt.

En del besvarelser antar at denne requesten krever innlogging. Det var ikke meningen (men ser jo at det er fornuftig). Det kan gis full score uten innlogging.

Løsningsforslag:

```
1 @Controller
  @RequestMapping("/nyansatt")
  public class NyAnsattController {

    1 @Autowired
      AnsattDbService ansattService;

    1 @GetMapping
      public String hentNyAnsattSkjema(Model model) {

    3         List<String> stillinger = ansattService.finnStillinger();
    3         model.addAttribute("stillinger", stillinger);
    1         return "nyansattskjema";
      }
  }
```

- d) Bruk skjelettet nedenfor for nyansattskjema.jsp og fullfør denne. Dvs. fyll inn det som mangler, merket med ?1? ... ?7?.

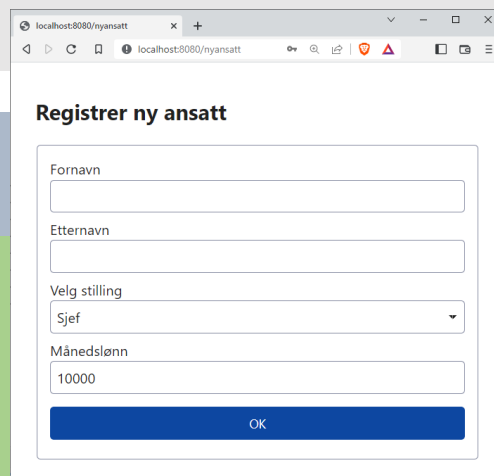
```
<html>
<body>
  <c:if test="${not empty nyansatt}"> //Satt ved registrering
    <p style="color:green;"> ?1? </p>
  </c:if>
  <c:if test="${not empty feilmelding}"> //Satt ved feil i registrering
    <p style="color:red;"> ?2? </p>
  </c:if>

  <h3>Registrer ny ansatt</h3>
  <form method="?3?">
    Fornavn <input type="?4?" name="fornavn"><br>
    Etternavn <input type="?5?" name="etternavn"><br>
    Velg stilling <select name="stilling">
      ?6? //Tips: Syntaks for et select-alternativ er
      //<option value="..."></option>
    </select><br>
    Månedslønn <input type="?7?" name="mndlonn"
      min="10000" max="99999" value="10000" ><br>
    <input type="submit" value="OK">
  </form>
</body>
</html>
```

Merknader til oppgave:

...

Løsningsforslag:



- 1 1: Ny ansatt lagt til!<br> \${nyansatt}
- 1 2: \${feilmelding}
- 1 3: post
- 1 4: text
- 1 5: text
- 2 6: <c:forEach var="s" items="\${stillinger}">
- 2     <option value="\${s}">\${s}</option></c:forEach>
- 1 7: number

- e) Fullfør controlleren fra c) med POST-mappingen som har ansvar for registrering / lagring av nyansatt. Hver request-parameter mottas separat, og må valideres. Hvis ikke alle data er gyldige er det greit nok med en generell feilmelding, se skjelett for oppgave d).

For å få full score må du skrive en hjelpeklasse `AnsattValidator` som har ansvar for input-valideringen. (Hvis du ikke rekker dette kan du bruke den som om den finnes.)

Valideringsregler:

- fornavn og etternavn inneholder minst 2 tegn hver
- månedslønn må være mellom 0 og 100 000
- valgt stilling må finnes i databasen
- Hvis ikke alle data er gyldige er det greit nok med en generell feilmelding, se skjelett for oppgave d)

Løsningsforslag se neste side: ...

```

@Controller
public class NyAnsattController {
    @Autowired private LogginnService ls;
    @Autowired private AnsattDbService ansattService;
    @Autowired private AnsattValidator validator;

    @PostMapping("/nyansatt")
    public String registrerNyAnsatt(
1        @RequestParam String fornavn, @RequestParam String etternavn,
1        @RequestParam String stilling, @RequestParam Integer mndlonn,
        HttpServletRequest request, RedirectAttributes ra) {

1        if (!ls.erLoggetInn(request)) {
1            return "redirect:innlogging.html";
        }

1        if (!erAllInputGyldig(fornavn, etternavn, stilling,
            ansattService.finnStilling(), mndlonn)) {

1            ra.addFlashAttribute("feilmelding",
                "Ikke all input var gyldig. Prøv på nytt.");
        } else {
2            Ansatt nyansatt = ansattService.registrerNyAnsatt(
                fornavn, etternavn, stilling, mndlonn);
1            ra.addFlashAttribute("nyansatt", nyansatt);
        }
2        return "redirect:nyansatt";
    }
}

//-----

@Service
public class AnsattValidator {

    public boolean erAllInputGyldig(String fornavn, String etternavn,
        String stilling, List<String> stillinger, int mndlonn) {

        return erGyldigNavn(fornavn) && erGyldigNavn(etternavn)
            && erGyldigStilling(stilling, stillinger)
            && erGyldigMndlonn(mndlonn);
    }

    public boolean erGyldigNavn(String navn) {
        return navn != null && navn.length() >= 2;
    }

    public boolean erGyldigStilling(String stilling, List<String> stillinger) {
        return stilling != null && stillinger != null
            && stillinger.contains(stilling);
    }

    public boolean erGyldigMndlonn(int mndlonn) {
        return mndlonn > 0 && mndlonn < 100_000;
    }
}

```