

# **EKSAMENSOPPGAVE**

**Bokmål**

**Emnekode: DAT108**

**Emnenavn: Programmering og webapplikasjoner**

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

**Dato: 10. juni 2024**

---

Eksamensform: Skriftlig skoleeksamen (Wiseflow | FLOWmulti)

Eksamenstid: 4 timer (0900-1300)

Antall eksamensoppgaver: 6

Antall sider (medregnet denne): 13

Antall vedlegg: Ingen

Tillatte hjelpemiddel: Ingen

Lærere:     Lars-Petter Helland (928 28 046)  
               Erlend Raa Vågset (57 72 26 08)  
               Bjarte Kileng (909 97 348)

**LYKKE TIL!**

## Oppgave 1 (10% ~ 24 minutter) – Strømmer

Flervalgsoppgave: Oppgavetekst i Wiseflow

## Oppgave 2 (10% ~ 24 minutter) – Funksjonelle kontrakter

Flervalgsoppgave: Oppgavetekst i Wiseflow

## Oppgave 3 (10% ~ 24 minutter) – JavaScript teori

Flervalgsoppgave: Oppgavetekst i Wiseflow



### Felles info for oppgave 4, 5 og 6

Du skal svare på disse i Wiseflow. Ved å bruke kodevedlegg vil dere få uthevet syntaks og korrekte innrykk. Vi anbefaler ett vedlegg per oppgave. For å sikre automatisk lagring, følg denne oppskriften.

- 1 - Start med å gi vedlegget et navn. For eksempel «Oppgave 4.java».
- 2 - Klikk lagre. Da aktiverer du automatisk lagring.
- 3 - Start å skrive kode

Pass på at du har svart på alle oppgaver før du prøver å få oppgaver perfekte. Hvis du mener noe er uklart i oppgaven, presiser forutsetningene dine i besvarelsen.

Oppgavene begynner på neste side.

## Oppgave 4 (15% ~ 36 minutter) – Tråder og trådsikkerhet

I denne oppgaven skal du jobbe med å simulere en iskremkiosk.

I iskremkiosken jobber det

- En selger som tar imot bestillinger fra kunder
- To iskremmakere som lager iskrem ut fra bestillingene
- To servitører som serverer ferdige iskremer til kundene

Kiosken holder orden på mottatte bestillinger og ferdige iskremer i to køer (én for bestillinger og én for ferdige iskremer).

Simuleringen gjøres ved at selger(e), iskremmaker(e) og servitør(er) legger inn og fjerner bestillinger og iskremer fra disse to køene i et visst tempo.

Koden vist nedenfor skisserer løsningen, men det mangler noe kode (som du skal legge inn), og løsningen har noen problemer (som du skal beskrive og løse).

```
class Selger implements Runnable {  
  
    private final Kiosk kiosk;  
  
    Selger(Kiosk kiosk) {  
        this.kiosk = kiosk;  
    }  
  
    @Override  
    public void run() {  
        while (true) {  
            kiosk.taImotBestilling();  
        }  
    }  
}
```

```
class Iskremmaker implements Runnable {  
  
    private final Kiosk kiosk;  
  
    Iskremmaker(Kiosk kiosk) {  
        this.kiosk = kiosk;  
    }  
  
    @Override  
    public void run() {  
        while (true) {  
            kiosk.lagIskrem();  
        }  
    }  
}
```

```
class Servitør implements Runnable {
```

```
    private final Kiosk kiosk;
```

```
    Servitør(Kiosk kiosk) {  
        this.kiosk = kiosk;  
    }
```

```
    @Override  
    public void run() {  
        while (true) {  
            kiosk.serverIskrem();  
        }  
    }  
}
```

```
class Kiosk {
```

```
    private final Queue<String> motatteBestillinger;  
    private final Queue<String> ferdigeIskremer;
```

```
    private int bestillingsTeller = 0;  
    private int iskremTeller = 0;
```

```
    Kiosk() {  
        motatteBestillinger = new LinkedList<>();  
        ferdigeIskremer = new LinkedList<>();  
    }
```

```
    // BESKRIVELSE:  
    // Simulerer at en selger tar imot en bestilling.
```

```
    public void taImotBestilling() {
```

```
        // Simulerer tiden det tar å ta imot en bestilling  
        Thread.sleep(500);
```

```
        // Øker telleren som teller antall bestillinger som er mottatt  
        bestillingsTeller++;
```

```
        String bestilling = "Bestilling " + bestillingsTeller;  
        System.out.println(bestilling);
```

```
        // Bestillingen er klar, og legges i kø for behandling  
        // API: Queue.offer(E e) - Legger inn elementet e bakerst i køen.  
        motatteBestillinger.offer(bestilling);  
    }
```

```

// BESKRIVELSE:
// Simulerer at en iskremmaker lager en is som er bestilt.

public void lagIskrem() {

    // En bestilling hentes ut, og fjernes fra køen
    // API: Queue.poll() - Henter ut og fjerner fremste element i køen,
    // eller returnerer null hvis køen er tom.
    String bestilling = motatteBestillinger.poll();

    // Simulerer tiden det tar å lage en is
    Thread.sleep(1000);

    // Øker telleren som teller antall is som har blitt lagd
    iskremTeller++;

    String iskrem = "Iskrem " + iskremTeller + " fra " + bestilling;
    System.out.println(iskrem);

    // Isen er nå klar til å serveres, og legges inn i iskremkøen
    ferdigeIskremer.offer(iskrem);
}

// BESKRIVELSE:
// Simulerer at en servitør serverer en iskrem.

public void serverIskrem() {

    // En is hentes ut, og fjernes fra køen
    String iskrem = ferdigeIskremer.poll();

    // Simulerer tiden det tar å servere en is
    Thread.sleep(300);

    // Isen er nå servert
    System.out.println("Serverer: " + iskrem);
}
}

```

```

public class IsMain {

    public static void main(String[] args) {

        Kiosk kiosk = new Kiosk();

        // a) Legg til kode for a) her. Se oppgaveteksten nedenfor.

        // b) Legg til kode for b) her. Se oppgaveteksten nedenfor.

    }
}

```

## Oppgaver

- a) Skriv koden i `main()` som oppretter
  - en tråd som simulerer en selger
  - to tråder som simulerer to iskremmakere
  - to tråder som simulerer to servitører
- b) Skriv koden i `main()` som starter de 5 trådene du opprettet i a)
- c) Forklar kort forskjellen på å skrive **`selger.start()`** og **`selger.run()`**.
- d) Klassen **Kiosk** bruker to køer (av typen `Queue`) implementert som kjedede lister (`LinkedList`) for å holde orden på mottatte bestillinger og iskrem som er klar til å serveres. Hvilke problemer kan vi få i simuleringen vår når vi velger å bruke `Queue` / `LinkedList` for køene? Hvordan kan vi fikse/forbedre koden slik at vi ikke får disse problemene? Du trenger ikke skrive kode, kun beskrive problemene og skissere mulige løsninger.
- e) Løsningen vår har også et annet problem, relatert til at vi har flere tråder som kjører samtidig. Gi en kort forklaring på hva som er galt, og beskriv kort hvordan dette kan fikses. Du trenger ikke skrive kode, kun beskrive problemene og skissere mulige løsninger. Hint: Se på tellerne.
- f) Gi en kort forklaring på hva en vranglås / dødlås (deadlock) er for noe. Du trenger ikke skrive kode.

## Oppgave 5 (15% ~ 36 minutter) – JavaScript programmering

En webside inneholder to felt. Det første feltet lar bruker registrere deltager i en konkurranse med sluttiden for deltager. Skjermbildet under viser hvordan dette feltet kan se ut:



Deltager registrering

Anne Annesen 23

Registrer resultat

*Skjermbilde 1: Felt for å registrere deltager*

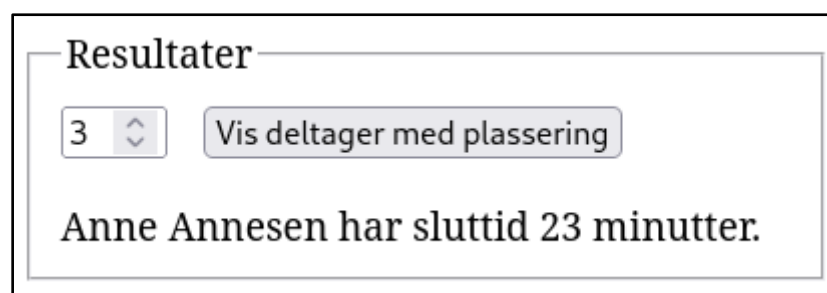
I Skjermbilde 1 blir deltager Anne Annesen registrert med en sluttid på 23 minutter. Deltager i dette skjermbildet kan representeres med et objekt på følgende form:

```
const deltager = {  
  'navn': 'Anne Annesen',  
  'tid': 23  
};
```

*Kode-eksempel 1: Anne Annesen med tiden 23 minutter*

Deltageren må lagres i en liste for senere bruk, f.eks. ved å bruke en forekomst av en JavaScript **Array**.

Det neste feltet lar bruker vise en deltager i konkurransen ved å søke på plassering. Skjermbildet under viser hvordan dette feltet kan se ut:



Resultater

3 Vis deltager med plassering

Anne Annesen har sluttid 23 minutter.

*Skjermbilde 2: Resultat for deltager på tredje plass*

I Skjermbilde 2 har bruker hentet fram deltager på tredje plass i konkurransen, som er Anne Annesen. Resultatlisten må altså være sortert med deltager som har korteste sluttid først.

Applikasjonen bruker HTML-koden under for å registre deltager:

```
<fieldset class="registrering" id="registrering">
  <legend>Deltager registrering</legend>

  <div>
    <input type="text" size="20" placeholder="Navn på deltager">
    <input type="number" size="11" placeholder="Tid i minutter">
    <button type="button">Registrer resultat</button>
  </div>
</fieldset>
```

*Kode-eksempel 2: HTML-kode for å registrere deltager*

Applikasjonen bruker HTML-koden under for å søke på deltager ut fra plassering i konkurransen:

```
<fieldset class="resultat" id="resultat">
  <legend>Resultater</legend>
  <div>
    <input type="number" size="3" value="1" min="1">
    <button>Vis deltager med plassering</button>
  </div>
  <p class="hidden plassering">
    <span></span> har sluttid <span></span> minutter.
  </p>
  <p class="plassholder">Her skal resultatet vises.</p>
</fieldset>
```

*Kode-eksempel 3: HTML-kode for å søke på deltager*



Applikasjonen inneholder følgende JavaScript-kode:

```
class KonkurranseKontroller {
  #liste = [];

  // Legg til her eventuelle flere private felt

  constructor(felt1ref, felt2ref) {
    this.#navnelement =
      felt1ref.querySelector("input[type='text']");
    this.#tidelement =
      felt1ref.querySelector("input[type='number']");
    const regbt = felt1ref.querySelector("button");
    regbt.addEventListener("click",
      () => { this.#regdeltager() }
    );

    // Legg inn kode her for felt2
  }

  #regdeltager() {
    // Legg inn kode her for å registrere deltager
  }

  // Legg til metod(er) for å kunne søke på deltager
}

const felt1 = document.getElementById("registrering");
const felt2 = document.getElementById("resultat");
new KonkurranseKontroller(felt1, felt2);
```

*Kode-eksempel 4: JavaScript-kode for å registrere og søke på deltager*

**Oppgave:** Fyll inn den manglende koden i JavaScript-klassen **KonkurranseKontroller**.

**Hjelp:**

- Kode-eksemplet under setter inn et nytt element i tabellen *this.#liste* mellom de eksisterende elementene med indeks 2 og indeks 3:

```
this.#liste.splice(3, 0, { 'navn': 'Anne Annesen', 'tid': 23 });
```

Det nye elementet får indeks 3 og alle påfølgende element øker sin indeks verdi med 1. Altså vil elementet som hadde indeks 3 få indeks 4.

- Kode-eksemplet under sorterer tabellen *this.#liste* stigende på sluttid til deltagerne:

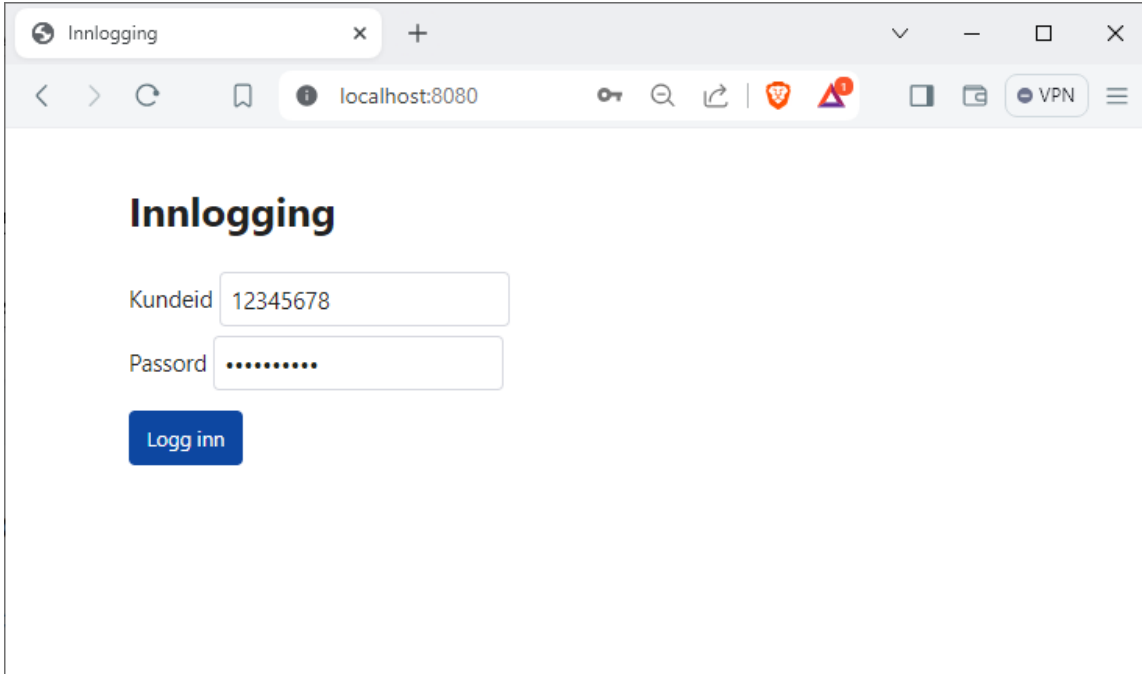
```
this.#liste.sort((d1, d2) => { return d1.tid > d2.tid });
```

- Både **Map**, **Set** og **Array** har en metode *forEach*.
- HTML input elementer sin attributt *value* er nåværende verdi til feltet.

## Oppgave 6 (40% ~ 96 minutter) – Web backend med Spring MVC

I denne oppgaven skal vi jobbe litt med en kontooversikt i en tenkt nettbank.

Vi tenker oss at vi logger oss inn med kunde-id og passord som vist i dette skjermbildet:



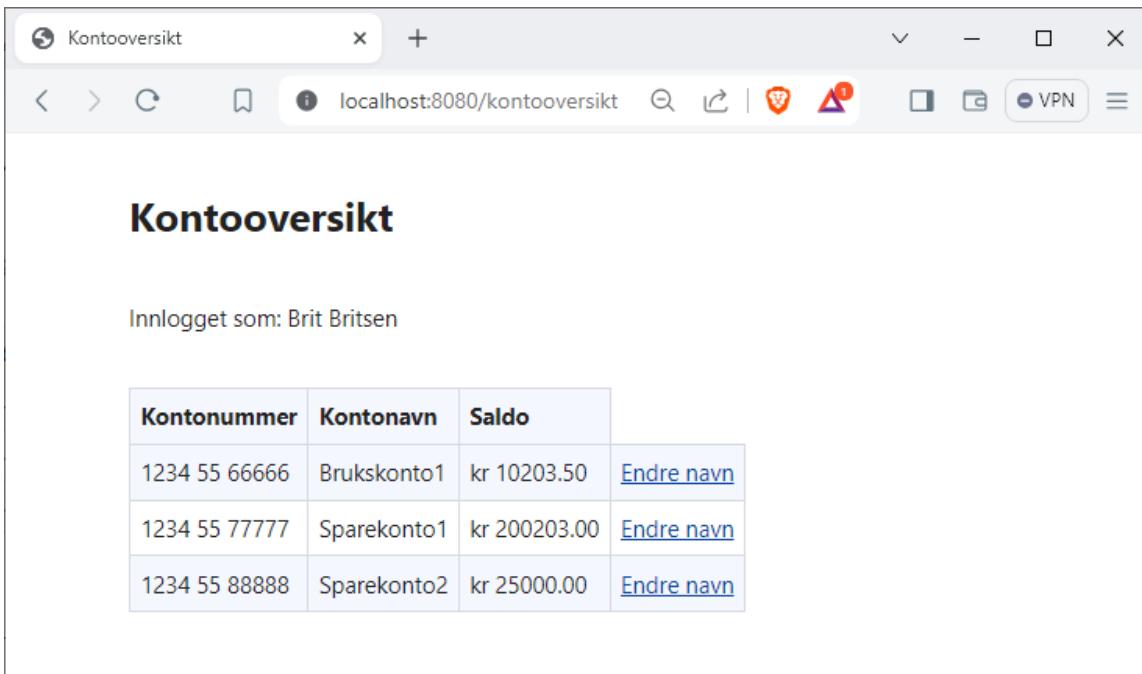
**Innlogging**

Kundeid

Passord

[Logg inn](#)

Etter vellykket innlogging kommer vi til kontooversikt:



**Kontooversikt**

Innlogget som: Brit Britsen

Kontonummer	Kontonavn	Saldo	
1234 55 66666	Brukskonto1	kr 10203.50	<a href="#">Endre navn</a>
1234 55 77777	Sparekonto1	kr 200203.00	<a href="#">Endre navn</a>
1234 55 88888	Sparekonto2	kr 25000.00	<a href="#">Endre navn</a>

Vi ser at vi er innlogget som Brit Britsen, og at vi har 3 kontoer med kontonummer, kontonavn og saldo.

Vi ser også at det er en lenke (<a href ...>) for hver konto som gir oss mulighet til å endre kontonavn for denne kontoen.

Hvis vi klikker på lenken for den øverste kontoen i eksempelet, kommer vi til dette skjermbildet:

Endre kontonavn

Innlogget som: Brit Britsen

Kontonummer	Kontonavn
1234 55 66666	Brukskonto1

Lagre nytt navn

Her kan vi taste inn ønsket nytt kontonavn, trykke på knappen «Lagre nytt navn», og kontoen blir oppdatert med nytt navn i databasen. Etter oppdatering kommer vi tilbake til kontooversikten igjen.

Forespørsler om kontooversikt, endre-kontonavn-side, og lagring av nytt kontonavn krever at du er innlogget. For enkelthets skyld kan alle ugyldige forespørsler videresendes til en statisk html-side **feilside.html**.

Oversikt over request-mappings, views og redirects for gyldige/normale forespørsler:

Beskrivelse	Request	Tilhørende View	Neste
Se innloggingsside	GET /logginn	logginn.jsp	---
Logge inn	POST /logginn	---	/kontooversikt
Se Kontooversikt	GET /kontooversikt	kontooversikt.jsp	---
Se Endre kontonavn	GET /endrekontonavn	endrekontonavn.jsp	---
Lagre nytt kontonavn	POST /endrekontonavn	---	/kontooversikt

## Litt om data og hjelpeklasser

Vi har lagret data om kunder og kontoer i en database, og henting og lagring av data gjøres via Spring-JpaRepositoriene **KundeRepo** og **KontoRepo**.

Java-entitetsklassen **Kunde** har disse egenskapene:

- int **id** (primærnøkkel)
- String **fornavn**
- String **etternavn**

Java-entitetsklassen **Konto** har disse egenskapene:

- String **kontonr** (primærnøkkel)
- String **kontonavn**
- BigDecimal **saldo**
- Kunde **kontoeier** (referanse til kunden som kontoen tilhører)

Du kan anta at disse klassene har de nødvendige get-metoder.

For enkelhets skyld tenker vi oss at alle requests blir håndtert av én Spring-controller kalt **BankController**.

For at BankController ikke skal trenge å forholde seg direkte til KundeRepo og KontoRepo, er det lagt inn en Spring-service **BankService** «oppå disse», som BankController kan bruke.

**BankService** har følgende metoder:

- Kunde **hentKunde**(int kundeid)
- Konto **hentKonto**(String kontonr)
- List<Konto> **hentAlleKontoerForKunde**(int kundeid)
- void **oppdaterKontoMedNyttNavn**(String kontonr, String nyttnavn)

Siden det er krav om innlogging, er det også laget en **LogginnService**, med følgende metoder:

- void **loggInn**(HttpServletRequest request, Kunde kunde)
- void **loggUt**(HttpServletRequest request)
- boolean **erLoggetInn**(HttpServletRequest request)
- Kunde **hentInnloggetKunde**(HttpServletRequest request)

Når bruker er innlogget vil et Kunde-objekt (uten liste av kontoer) for denne kunden være lagret som attributtet «kunde» i sesjonen.

## Oppgaver

- a) Skriv en metode **henteKontooversikt(...)** i BankController, som håndterer forespørsel om å se Kontooversikt. Du tar utgangspunkt i at BankController er satt opp slik:

```
@Controller
public class BankController {

    @Autowired LogginnService logginnService;
    @Autowired BankService bankService;
    ...
}
```

Husk å sjekke at bruker er innlogget. Kontoene skal hentes på nytt fra databasen for hver forespørsel om å se kontooversikten (data kan være oppdatert siden sist).

- b) Skriv viewet for kontooversikten, **kontooversikt.jsp**. Du trenger kun å skrive <body>-delen.

Du blir ikke trukket om du ikke bruker tabell.

Du får poeng om du skriver teksten «Ingen aktive kontoer» i stedet for en tom tabell hvis innlogget kunde ikke har noen kontoer.

- c) Skriv en metode **endreKontonavnSide(...)** i BankController, som håndterer forespørsel om å se endre-kontonavn-siden.

I tillegg til å sjekke om bruker er innlogget skal du her også sjekke at kunden eier den kontoen som skal endre navn. Tips: Lag gjerne en hjelpemetode for dette, f.eks. **boolean erGyldigKontoForInnloggetKunde(HttpServletRequest request, String kontonr)**. Denne kan også gjenbrukes i oppgave e).

- d) Skriv viewet for endre-kontonavn-siden, **endrekontonavn.jsp**. Du trenger kun å skrive <body>-delen.

- e) Skriv en metode **lagreNyttKontonavn(...)** i BankController, som håndterer forespørsel om å endre navnet på en konto, altså foreta endringen i databasen.

I tillegg til å sjekke om bruker er innlogget skal du også her sjekke at kunden eier den kontoen som skal endre navn. Sjekk også at kontonavn er gyldig, f.eks. mellom 2-20 tegn.

- f) Skriv metoden **oppdaterKontoMedNyttNavn(...)** i BankService. Du tar utgangspunkt i at BankService er satt opp slik:

```
@Service
public class BankService {

    @Autowired KundeRepo kundeRepo;
    @Autowired KontoRepo kontoRepo;
    ...
}
```

# EKSAMENSOPPGÅVE

Nynorsk

**Emnekode: DAT108**

**Emnenamn: Programmering og webapplikasjoner**

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

**Dato: 10. juni 2024**

---

Eksamensform: Skriftleg skuleeksamen (Wiseflow | FLOWlock)

Eksamenstid: 4 timer (0900-1300)

Tal på eksamensoppgåver: 6

Tal på sider (medrekna denne): 13

Tal på vedlegg: Ingen

Tillate hjelpemiddel: Ingen

Lærarar:    Lars-Petter Helland (928 28 046)  
              Erlend Raa Vågset (57 72 26 08)  
              Bjarte Kileng (909 97 348)

**LUKKE TIL!**

## Oppgave 1 (10% ~ 24 minutt) – Straumar

Fleirvalsoppgåve: Oppgåvetekst i Wiseflow

## Oppgave 2 (10% ~ 24 minutt) – Funksjonelle kontraktar

Fleirvalsoppgåve: Oppgåvetekst i Wiseflow

## Oppgave 3 (10% ~ 24 minutt) – JavaScript teori

Fleirvalsoppgåve: Oppgåvetekst i Wiseflow

∞∞∞

## Felles info for oppgave 4, 5 og 6

Du skal svare på desse i Wiseflow. Ved å bruke kodevedlegg vil de få utheva syntaks og korrekte innrykk. Vi tilrår eitt vedlegg per oppgåve. For å sikre automatisk lagring, følg denne oppskrifta.

- 1 - Start med å gi vedlegget eit namn. Til dømes «Oppgave 4.java».
- 2 - Klikk lagre. Då aktiverer du automatisk lagring.
- 3 - Start å skrive kode

Pass på at du har svart på alle oppgåver før du prøver å få oppgåver perfekte. Viss du meiner noko er uklart i oppgåva, presiser føresetnadene dine i svaret.

Oppgåvene byrjar på neste side.

## Oppgave 4 (15% ~ 36 minutt) – Trådar og trådtryggleik

I denne oppgåva skal du jobbe med å simulere ein iskremkiosk.

I iskremkiosken jobbar det

- Ein seljar som tar imot bestillingar frå kundar
- To iskremmakarar som lager iskrem ut frå bestillingane
- To servitørar som serverer ferdige iskremar til kundane

Kiosken held orden på mottekne bestillingar og ferdige iskremar i to køar (éin for bestillingar og éin for ferdige iskremar).

Simuleringa blir gjord ved at seljar(ar), iskremmakar(ar) og servitør(ar) legger inn og fjernar bestillingar og iskremar frå desse to køane i eit visst tempo.

Koden vist nedanfor skisserer løysinga, men det manglar noko kode (som du skal leggje inn), og løysinga har nokre problem (som du skal beskrive og løyse).

```
class Selger implements Runnable {  
  
    private final Kiosk kiosk;  
  
    Selger(Kiosk kiosk) {  
        this.kiosk = kiosk;  
    }  
  
    @Override  
    public void run() {  
        while (true) {  
            kiosk.taImotBestilling();  
        }  
    }  
}
```

```
class Iskremmaker implements Runnable {  
  
    private final Kiosk kiosk;  
  
    Iskremmaker(Kiosk kiosk) {  
        this.kiosk = kiosk;  
    }  
  
    @Override  
    public void run() {  
        while (true) {  
            kiosk.lagIskrem();  
        }  
    }  
}
```



```
class Servitør implements Runnable {
```

```
    private final Kiosk kiosk;
```

```
    Servitør(Kiosk kiosk) {  
        this.kiosk = kiosk;  
    }
```

```
    @Override  
    public void run() {  
        while (true) {  
            kiosk.serverIskrem();  
        }  
    }  
}
```

```
class Kiosk {
```

```
    private final Queue<String> motatteBestillinger;  
    private final Queue<String> ferdigeIskremer;
```

```
    private int bestillingsTeller = 0;  
    private int iskremTeller = 0;
```

```
    Kiosk() {  
        motatteBestillinger = new LinkedList<>();  
        ferdigeIskremer = new LinkedList<>();  
    }
```

```
    // BESKRIVELSE:  
    // Simulerer at en selger tar imot en bestilling.
```

```
    public void taImotBestilling() {
```

```
        // Simulerer tiden det tar å ta imot en bestilling  
        Thread.sleep(500);
```

```
        // Øker telleren som teller antall bestillinger som er mottatt  
        bestillingsTeller++;
```

```
        String bestilling = "Bestilling " + bestillingsTeller;  
        System.out.println(bestilling);
```

```
        // Bestillingen er klar, og legges i kø for behandling  
        // API: Queue.offer(E e) - Legger inn elementet e bakerst i køen.  
        motatteBestillinger.offer(bestilling);  
    }
```

```

// BESKRIVELSE:
// Simulerer at en iskremmaker lager en is som er bestilt.

public void lagIskrem() {

    // En bestilling hentes ut, og fjernes fra køen
    // API: Queue.poll() - Henter ut og fjerner fremste element i køen,
    // eller returnerer null hvis køen er tom.
    String bestilling = motatteBestillinger.poll();

    // Simulerer tiden det tar å lage en is
    Thread.sleep(1000);

    // Øker telleren som teller antall is som har blitt lagd
    iskremTeller++;

    String iskrem = "Iskrem " + iskremTeller + " fra " + bestilling;
    System.out.println(iskrem);

    // Isen er nå klar til å serveres, og legges inn i iskremkøen
    ferdigeIskremer.offer(iskrem);
}

// BESKRIVELSE:
// Simulerer at en servitør serverer en iskrem.

public void serverIskrem() {

    // En is hentes ut, og fjernes fra køen
    String iskrem = ferdigeIskremer.poll();

    // Simulerer tiden det tar å servere en is
    Thread.sleep(300);

    // Isen er nå servert
    System.out.println("Serverer: " + iskrem);
}
}

```

```

public class IsMain {

    public static void main(String[] args) {

        Kiosk kiosk = new Kiosk();

        // a) Legg til kode for a) her. Se oppgaveteksten nedenfor.

        // b) Legg til kode for b) her. Se oppgaveteksten nedenfor.

    }
}

```

## Oppgåver

- a) Skriv koden i `main()` som opprettar
  - ein tråd som simulerer en seljar
  - to trådar som simulerer to iskremmakarar
  - to trådar som simulerer to servitørar
- b) Skriv koden i `main()` som startar dei 5 trådane du oppretta i a)
- c) Forklar kort forskjellen på å skrive **`selger.start()`** og **`selger.run()`**.
- d) Klassen **Kiosk** bruker to køar (av typen `Queue`) implementert som kjeda lister (`LinkedList`) for å holde orden på mottekne bestillingar og iskrem som er klar til å serverast. Kva problem kan vi få i simuleringa vår når vi vel å bruke `Queue` / `LinkedList` for køane? Korleis kan vi fikse/forbetre koden slik at vi ikkje får desse problema? Du treng ikkje skrive kode, berre beskrive problema og skissere moglege løysingar.
- e) Løysinga vår har også eit anna problem, relatert til at vi har fleire trådar som køyrer samtidig. Gi ei kort forklaring på kva som er gale, og beskriv kort korleis dette kan fiksast. Du treng ikkje skrive kode, berre beskrive problema og skissere moglege løysingar. Hint: Se på teljarane.
- f) Gi ei kort forklaring på kva ein vranglås / dødlås (deadlock) er for noko. Du treng ikkje skrive kode.

## Oppgave 5 (15% ~ 36 minutt) – JavaScript programmering

Ein webside inneheld to felt. Det første feltet lar brukar registrera deltakarar i ein konkurranse med sluttid for deltakar. Skjermbiletet under viser korleis dette feltet kan sjå ut:



*Skjermbilete 1: Felt for å registrere deltakar*

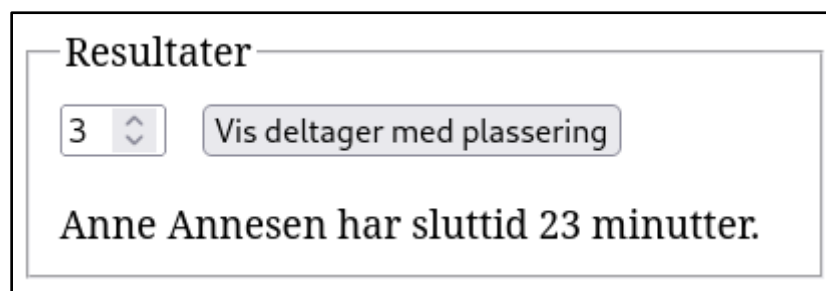
I Skjermbilete 1Skjermbilde 1 verte deltakar Anne Annesen registrert med ei sluttid på 23 minutt. Deltakar i dette skjermbiletet kan verte representert med eit objekt på fylgjande form:

```
const deltager = {  
  'navn': 'Anne Annesen',  
  'tid': 23  
};
```

*Kode døme 1: Anne Annesen med tid 23 minutt*

Deltakaren må verte lagra i ei liste for seinare bruk, til dømes ved å nytte ein førekomst av ein JavaScript **Array**.

Det neste feltet lar brukar vise ein deltakar i konkurransen ved å søkje på plassering. Skjermbiletet under viser korleis dette feltet kan sjå ut:



*Skjermbilete 2: Resultat for deltakar på tredje plass*

I Skjermbilete 2 har bruker henta fram deltakar på tredje plass i konkurransen, som er Anne Annesen. Resultatlista må altså være sortert med deltakar som har kortaste sluttid først.

Applikasjonen nyttar HTML-koden under for å registrera deltakar:

```
<fieldset class="registrering" id="registrering">
  <legend>Deltager registrering</legend>

  <div>
    <input type="text" size="20" placeholder="Navn på deltager">
    <input type="number" size="11" placeholder="Tid i minutter">
    <button type="button">Registrer resultat</button>
  </div>
</fieldset>
```

*Kode dørne 2: HTML-kode for å registrera deltakar*

Applikasjonen nyttar HTML-koden under for å søkje på deltakar ut frå plassering i konkurransen:

```
<fieldset class="resultat" id="resultat">
  <legend>Resultater</legend>
  <div>
    <input type="number" size="3" value="1" min="1">
    <button>Vis deltager med plassering</button>
  </div>
  <p class="hidden plassering">
    <span></span> har sluttid <span></span> minutter.
  </p>
  <p class="plassholder">Her skal resultatet vises.</p>
</fieldset>
```

*Kode dørne 3: HTML-kode for å søkje på deltakar*

Applikasjonen inneheld fylgjande JavaScript-kode:

```
class KonkurranseKontroller {
  #liste = [];

  // Legg til her eventuelle fleire private felt

  constructor(felt1ref, felt2ref) {
    this.#navnelement =
      felt1ref.querySelector("input[type='text']");
    this.#tidelement =
      felt1ref.querySelector("input[type='number']");
    const regbt = felt1ref.querySelector("button");
    regbt.addEventListener("click",
      () => { this.#regdeltager() }
    );

    // Legg inn kode her for felt2
  }

  #regdeltager() {
    // Legg inn kode her for å registrera deltakar
  }

  // Legg til metode(r) for å kunne søkje på deltakar
}

const felt1 = document.getElementById("registrering");
const felt2 = document.getElementById("resultat");
new KonkurranseKontroller(felt1, felt2);
```

*Kode døme 4: JavaScript-kode for å registrera og søke på deltakar*

**Oppgåve:** Fyll inn den manglande koden i JavaScript-klassa **KonkurranseKontroller**.

**Hjelp:**

- Døme nedanfor med kode sett inn eit nytt element i tabellen *this.#liste* mellom dei eksisterande elementa med indeks 2 og indeks 3:

```
this.#liste.splice(3, 0, { 'navn': 'Anne Annesen', 'tid': 23 });
```

Det nye elementet får indeks 3 og alle påfølgjande element aukar sitt indeks verdi med 1. Altså vil elementet som hadde indeks 3 få indeks 4.

- Døme nedanfor med kode sorterer tabellen *this.#liste* stigande på sluttid til deltakarane:

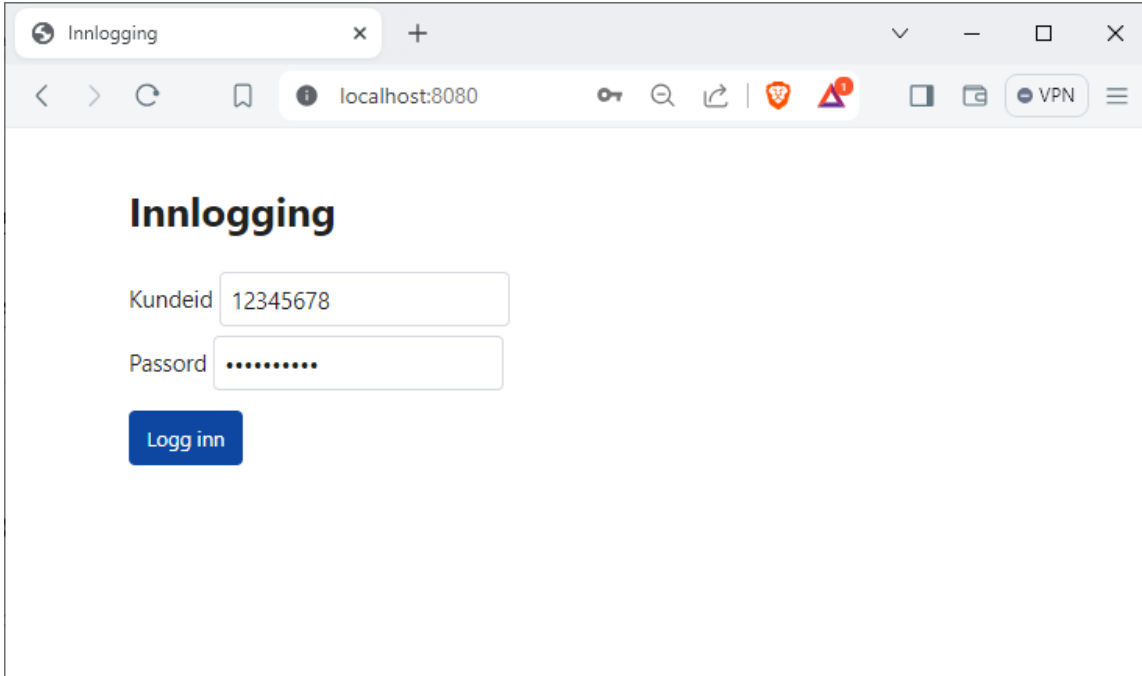
```
this.#liste.sort((d1, d2) => { return d1.tid > d2.tid });
```

- Både **Map**, **Set** og **Array** har ein metode *forEach*.
- HTML input element sin attributt *value* er nåverande verdi til feltet.

## Oppgave 6 (40% ~ 96 minutt) – Web backend med Spring MVC

I denne oppgåva skal vi jobbe litt med ei kontooversikt i ein tenkt nettbank.

Vi tenkjer oss at vi logger oss inn med kunde-id og passord som vist i dette skjermbiletet:



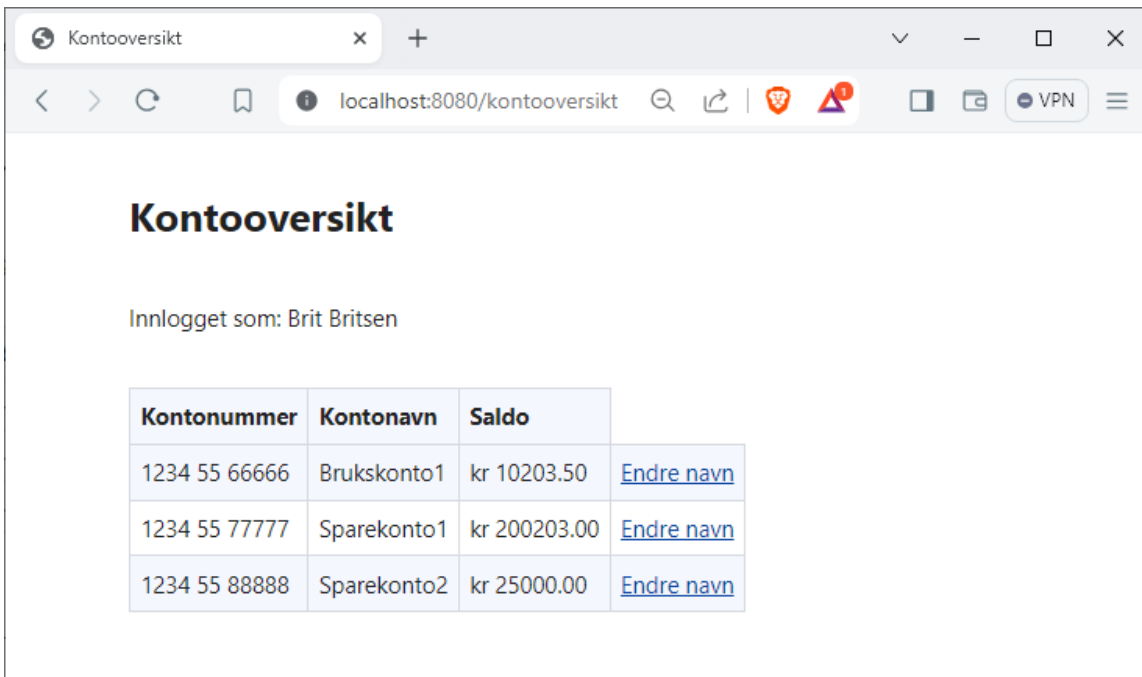
**Innlogging**

Kundeid

Passord

[Logg inn](#)

Etter vellykka innlogging kjem vi til kontooversikt:



**Kontooversikt**

Innlogget som: Brit Britsen

Kontonummer	Kontonavn	Saldo	
1234 55 66666	Brukskonto1	kr 10203.50	<a href="#">Endre navn</a>
1234 55 77777	Sparekonto1	kr 200203.00	<a href="#">Endre navn</a>
1234 55 88888	Sparekonto2	kr 25000.00	<a href="#">Endre navn</a>

Vi ser at vi er innlogga som Brit Britsen, og at vi har 3 kontoar med kontonummer, kontonamn og saldo.

Vi ser også at det er ei lenke (<a href ...>) for kvar konto som gir oss høve til å endre kontonamn for denne kontoen.

Viss vi klikkar på lenkja for den øvste kontoen i dømet, kjem vi til dette skjermbiletet:

Endre kontonavn

Innlogget som: Brit Britsen

Kontonummer	Kontonavn
1234 55 66666	Brukskonto1

Lagre nytt navn

Her kan vi taste inn ønsket nytt kontonamn, trykke på knappen «Lagre nytt navn», og kontoen blir oppdatert med nytt namn i databasen. Etter oppdatering kjem vi tilbake til kontooversikta igjen.

Førespurnader om kontooversikt, endre-kontonavn-side, og lagring av nytt kontonamn krev at du er innlogga. For enkelthets skyld kan alle ugyldige førespurnader vidaresendast til ein statisk html-side **feilside.html**.

Oversikt over request-mappings, views og redirects for gyldige/normale førespurnader:

Skildring	Request	Tilhøyrande View	Neste
Sjå Innloggingsside	GET /logginn	logginn.jsp	---
Logge inn	POST /logginn	---	/kontooversikt
Sjå Kontooversikt	GET /kontooversikt	kontooversikt.jsp	---
Sjå Endre kontonavn	GET /endrekontonavn	endrekontonavn.jsp	---
Lagre nytt kontonavn	POST /endrekontonavn	---	/kontooversikt



## Litt om data og hjelpeklassar

Vi har lagra data om kundar og kontoar i ein database, og henting og lagring av data blir gjort via Spring-JpaRepositoriene **KundeRepo** og **KontoRepo**.

Java-entitetsklassen **Kunde** har desse eigenskapane:

- int **id** (primærnøkkel)
- String **fornavn**
- String **etternavn**

Java-entitetsklassen **Konto** har disse eigenskapane:

- String **kontonr** (primærnøkkel)
- String **kontonavn**
- BigDecimal **saldo**
- Kunde **kontoeier** (referanse til kunden som kontoen høyrer til)

Du kan anta at desse klassene har de nødvendige get-metodar.

For enkelheits skuld tenkjer vi oss at alle requests blir handtert av éin Spring-controller kalla **BankController**.

For at BankController ikkje skal trenge å forholde seg direkte til KundeRepo og KontoRepo, er det lagt inn ein Spring-service **BankService** «oppå desse», som BankController kan bruke.

**BankService** har følgjande metodar:

- Kunde **hentKunde**(int kundeid)
- Konto **hentKonto**(String kontonr)
- List<Konto> **hentAlleKontoerForKunde**(int kundeid)
- void **oppdaterKontoMedNyttNavn**(String kontonr, String nyttnavn)

Sidan det er krav om innlogging, er det også laga ein **LogginnService**, med følgjande metodar:

- void **loggInn**(HttpServletRequest request, Kunde kunde)
- void **loggUt**(HttpServletRequest request)
- boolean **erLoggetInn**(HttpServletRequest request)
- Kunde **hentInnloggetKunde**(HttpServletRequest request)

Når bruker er innlogga vil eit Kunde-objekt (utan liste av kontoar) for denne kunden vere lagra som attributtet «kunde» i sesjonen.

## Oppgåver

- a) Skriv ein metode **henteKontooversikt(...)** i BankController, som handterer førespurnad om å sjå Kontooversikt. Du tek utgangspunkt i at BankController er sett opp slik:

```
@Controller
public class BankController {

    @Autowired LogginnService logginnService;
    @Autowired BankService bankService;
    ...
}
```

Husk å sjekke at brukar er innlogga. Kontoane skal hentast på nytt frå databasen for kvar førespurnad om å sjå kontooversikta (data kan vera oppdatert sidan sist).

- b) Skriv viewet for kontooversikta, **kontooversikt.jsp**. Du treng berre å skrive <body>-delen.

Du blir ikkje trekt om du ikkje bruker tabell.

Du får poeng om du skriv teksten «Ingen aktive kontoar» i staden for ein tom tabell viss innlogga kunde ikkje har nokon kontoar.

- c) Skriv ein metode **endreKontonavnSide(...)** i BankController, som handterer førespurnad om å sjå endre-kontonavn-siden.

I tillegg til å sjekke om brukar er innlogga skal du her også sjekke at kunden eig den kontoen som skal endre namn. Tips: Lag gjerne en hjelpemetode for dette, t.d. **boolean erGyldigKontoForInnloggetKunde(HttpServletRequest request, String kontonr)**. Denne kan også gjenbrukast i oppgåve e).

- d) Skriv viewet for endre-kontonavn-siden, **endrekontonavn.jsp**. Du treng berre å skrive <body>-delen.

- e) Skriv ein metode **lagreNyttKontonavn(...)** i BankController, som handterer førespurnad om å endre namnet på ein konto, altså foreta endringa i databasen.

I tillegg til å sjekke om brukar er innlogga skal du også her sjekke at kunden eig den kontoen som skal endre namn. Sjekk også at kontonamn er gyldig, t.d. mellom 2-20 teikn.

- f) Skriv metoden **oppdaterKontoMedNyttNavn(...)** i BankService. Du tek utgangspunkt i at BankService er sett opp slik:

```
@Service
public class BankService {

    @Autowired KundeRepo kundeRepo;
    @Autowired KontoRepo kontoRepo;
    ...
}
```