

## **LØSNINGSFORSLAG**

30.12.2020

m/merknader fra sensur

**Emnekode: DAT108**

**Emnenavn: Programmering og webapplikasjoner**

**Utdanning/kull/klasse: 2. klasse data/inf**

**Dato: 7. desember 2020**

---

Eksamensform: Skriftlig hjemmeksamen (Wiseflow | FLOWassign)

Eksamenstid: 4 timer (0900-1300) + 0,5 timer ekstra for innlevering

Antall eksamensoppgaver: 5

Antall sider (medregnet denne): 10

Antall vedlegg: Ingen

Tillatte hjelpemiddel: Alle.

Lærere:     Lars-Petter Helland  
               Bjarte Kileng

## Oppgave 1 (16% ~ 38 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) Gitt lambdauttrykket `s -> s.toLowerCase()`. Skriv en setning der du tilordner dette til en variabel. Husk å angi hvilken datatype variabelen er. Gi variabelen et passende navn. Tenk deg så at du har en liste av navn (av typen `List<String>`). Skriv kode, der variabelen over blir brukt, som gir utskrift med små bokstaver av alle navnene i listen, ett navn per linje.

Løsningsforslag:

```
Function<String, String> tilSmaaBokstaver = s -> s.toLowerCase();
navneliste.stream().map(tilSmaaBokstaver).forEach(System.out::println);
```

Alternativ løsning:

```
Function<String, String> tilSmaaBokstaver = s -> s.toLowerCase();
navneliste.forEach(navn -> System.out.println(tilSmaaBokstaver.apply(navn)));
```

Enda en alternativ løsning:

```
Function<String, String> tilSmaaBokstaver = s -> s.toLowerCase();
for (String navn : navneliste) {
    System.out.println(tilSmaaBokstaver.apply(navn));
}
```

Enda en alternativ løsning (som endrer innhold i opprinnelig liste):

```
UnaryOperator<String> tilSmaaBokstaver = s -> s.toLowerCase();
navneliste.replaceAll(tilSmaaBokstaver);
navneliste.forEach(System.out::println);
```

Merknad: `Function<String, String>` kan erstattes med `UnaryOperator<String>` i alle tilfellene.

Merknad: Det kan nok føles som å "gå over bekken etter vann" og meningsløst når definisjon og bruk av funksjonsvariabelen er i to påfølgende setninger. Vi må kanskje tenke oss at disse tingene foregår ulike steder i programmet, kanskje i ulike metoder til og med, der funksjonen er brukt som parameter.

- b) Gitt lambdauttrykket `tall -> tall % x == 0`. Skriv en metode med `x` som parameter som returnerer dette lambdauttrykket. Husk å få med returtype. Gi metoden et passende navn. Tenk deg så at du har en liste av heltall (av typen `List<Integer>`). Skriv kode, der metoden over blir brukt, som gir utskrift av alle tallene i listen som er delelige med 3, ett tall per linje.

Løsningsforslag, del i:

```
public static Predicate<Integer> erDeleligMed(int x) {  
    return tall -> tall % x == 0;  
}
```

Løsningsforslag, del ii:

```
listeAvTall.stream().filter(erDeleligMed(3)).forEach(System.out::println);
```

Alternativ løsning, del ii, der man skriver sitt eget "filter", f.eks. slik. Dette gir ikke full score:

```
for (int i : listeAvTall) {  
    if (erDeleligMed(3).test(i)) {  
        System.out.println(i);  
    }  
}
```

Tenk at vi har en liste med passord i klartekst som vi ønsker å analysere styrken på, samt en liste med de 10 mest vanlige passordene:

```
List<String> passordlisten = Arrays.asList(
    "qwerty", "123", "password", "peace&love", "abc", "12345678", "admin",
    "tomee", "fotball", "hei på deg");

List<String> tiMestVanligePassord = Arrays.asList(
    "123456", "123456789", "qwerty", "password", "1234567", "12345678",
    "12345", "iloveyou", "111111", "123123");
```

- c) Bruk streams til å lage en ny liste med de passordene i **passordlisten** som også finnes i listen over **tiMestVanligePassord**. (Tips: List har en metode `cointains()`).

Løsningsforslag:

```
List<String> vanligePassord = passordlisten.stream()
    .filter(tiMestVanligePassord::contains)
    .collect(Collectors.toList());
```

Merknad: Man bør kanskje også ha med `distinct()` for å fjerne duplikater og/eller bruke `Set<String>` og `Collectors.toSet()` for resultatet ...

IKKE en gyldig løsning på oppgaven, men kanskje den "enkleste" måten å gjøre det i praksis. Endrer innhold i opprinnelig liste. Er heller ikke støttet av alle List-implementasjoner (kan få `UnsupportedOperationException`):

```
passordlisten.retainAll(tiMestVanligePassord);
```

- d) Bruk streams til å beregne hva som er gjennomsnittlig passordlengde for passordene i **passordlisten**. (Tips: Vær litt obs på dette med Optional).

Løsningsforslag:

```
double snittlengde = passordlisten.stream()
    .map(String::length)
    .reduce(0, Integer::sum)          // eller ..., (sum, a) -> sum + a)
    / (double) passordlisten.size(); // -> 0/0.0 = NaN ved tom liste
```

Alternativ løsning med mapToInt() og IntStream (vi har ikke sett mye på dette, men det er vel denne som dukker opp ved søk på nettet):

```
double snittlengde = passordlisten.stream()
    .mapToInt(String::length)
    .average()
    .orElse(0.0); // -> 0.0 ved tom liste
//evt. .orElseThrow(); -> ...
//el. .getAsDouble(); -> ...
```

Enda en mulig løsning med en Collector som gjør gjennomsnitt:

```
double snittlengde = passordlisten.stream()
    .collect((Collectors.averagingInt(String::length)));
```

## Oppgave 2 (8% ~ 20 minutter) – Passordsikkerhet

Anta at en angriper har full tilgang til passord-dataene, dvs. hash og salt for alle brukernes passord, og at hashing-algoritmen er kjent.

Vi snakker om tre ulike typer angrep man da kan gjøre for å finne brukeres passord.

1. Brute force-angrep
2. Ordlisteangrep
3. Tabelloppslag-angrep

En av sikkerhetsmekanismene vi bruker når vi hasher passord er **salting**. Hvilke(n) av de tre angrepstypene hjelper salting mot? Hvordan? Begrunn svaret. Kan salting også hjelpe mot en problematikk uavhengig av angrepsmåte?

### Løsningsforslag:

#### Brute force-angrep

Salting hjelper ikke mye mot dette. Den som prøver å knekke passordene legger bare til saltet før hashing i hvert forsøk. Hvis antall passord som skal knekkes er  $n$  kan mangel på salting i teorien redusere tiden med en faktor  $n$  (kan gjenbruke beregninger på alle passordene).

#### Ordlisteangrep

Salting hjelper heller ikke mye mot dette. Ordlisteangrep er egentlig bare en litt lurere måte å prøve seg frem enn brute force. Hvis antall passord som skal knekkes er  $n$  kan mangel på salting i teorien redusere tiden med en faktor  $n$  (kan gjenbruke beregninger på alle passordene).

#### Tabelloppslag-angrep

Oppslagstabeller inneholder mengder av ferdig usaltede! hashede passord. Ved å salte med tilfeldig individuelt salt kan ikke passord forhånds-hashes på denne måten. Det er rett og slett alt for mange mulige salt-kombinasjoner. Salting gir en god beskyttelse mot denne typen angrep!

#### Generelt / uavhengig

Salting med individuelt salt gjør at to ulike brukere med samme passord vil få ulik hash. Dette er gunstig uansett hvilket angrep man bruker. Den som vil knekke passord må ta for seg hvert eneste passord, og har ikke nytte av å lagre beregnede hashverdier for senere oppslag.

**Merknad:** En del besvarelser sier at hashing av et langt passord tar lengre tid enn et kort, og bruker dette som et argument for salting. Det går for langt å ta dette i detalj her, men i praksis har dette liten betydning. Og har man iterasjoner av hashing (key stretching) er det kun input-0 som er selve passordet (m/ evt. salt). Deretter vil det jo være forrige hash (f.eks. 256 bit) som er input. ...

### Oppgave 3 (20% ~ 48 minutter) – JavaScript

- a) Nedenfor ser du JavaScript kode for to metoder som begge oppretter et nytt HTML P-element (tagg **P**) og legger inn noe data i P-elementet.

Metoden *showUserdata*:

```
function showUserdata(userData, rootElement) {  
    rootElement.insertAdjacentHTML('beforeend', '<p>${userData}</p>`'  
}
```

Metoden *viewUserdata*:

```
function viewUserdata(userData, rootElement) {  
    rootElement.insertAdjacentHTML('beforeend', '<p></p>`'  
    rootElement.lastElementChild.textContent = userData  
}
```

- i. Gå gjennom koden linje for linje og forklar koden.

Løsningsforslag:

Metode *showUserdata*:

Metoden *insertAdjacentHTML* setter inn som siste barn av *rootElement* et HTML **P**-element med innholdet i parameter *userData*.

Metode *viewUserdata*:

Metoden *insertAdjacentHTML* setter inn som siste barn av *rootElement* et tomt HTML **P**-element. Neste linje legger inn et tekstelement fra innholdet i *userData* inne i **P**-elementet.

- ii. Den ene av de to metoden over kan gi problemer i praksis. Diskuter dette og vis med et eksempel hvorfor denne metoden ikke bør brukes.

Løsningsforslag:

I metoden *showUserdata* vil *userData* bli tolket som HTML. Dette kan brukes av en uærlig person til å endre innholdet på siden.

I metoden *viewUserdata* vil *userData* bli satt inn som tekst. Eventuell HTML-kode blir ufarliggjort og vises som tekst.

forts. neste side ...

Eksempelet nedenfor bruker HTML og CSS for å skjule alt innholdet på websiden, og i stedet vise teksten i *userdata* sitt **SPAN** element:

```
const userdata = ``
showUserdata (userData)
```



- b) Begge kodelinjene under vil opprette en liste av HTML-elementer med tagg **DIV**.

```
const divElementList = rootElement.querySelectorAll('div')
const divElements = rootElement.getElementsByTagName('div')
```

Begge listene vil i utgangspunktet returnere de samme elementene, men vil kunne avvike etter hvert som JavaScript-kode arbeider med dokumentet.

- i. Utdyp forskjellen på de to listene, og vis eksempel på kode som gjør at de to listene ikke lengre inneholder de samme elementene.

Løsningsforslag:

Metoden *querySelectorAll* returnerer en statisk liste. Innholdet er **DIV**-elementene under *rootElement* som fantes da metoden ble kalt.

Metoden *getElementsByTagName* returnerer en dynamisk liste. Listen blir oppdatert automatisk når websiden oppdateres, og vil til enhver tid være en list av alle **DIV**-elementer under *rootElement*.

Koden under legger til et nytt DIV-element under *rootElement*. Listen *divElements* vil inneholde det nye elementet, men ikke listen *divElementList*.

```
rootElement.insertAdjacentHTML('beforeend', '<div>Hei</div>')
```

- ii. Legg til et HTML *class* attributt *info* på alle elementene i listen *divElementList*. Observer, elementene kan allerede ha andre *class* attributt elementer, og disse skal ikke røres.

Løsningsforslag:

```
divElementList.forEach(
  divElement => {divElement.classList.add('info')}
)
```

- c) Opprett en JavaScript klasse **Calculator**. Klassen har en metode *calculate* og to egenskaper *status* og *result* som kan brukes utenfor klassen (brukes **public**).

Metode *calculate* er gitt av følgende kode:

```
/**
 * Metode for å utføre utregning
 * @public
 * @param {String} operation - Matematikk operasjon
 * @param {Array.<String>} numberList - Array av input-data
 */
calculate(operation, numberList) {
  /* JavaScript kode for å utføre utregning */
}
```

Parameter *operation* angir en regneoperasjon, og parameter *numberList* er en **Array** av heltall. Listen av heltall kan ha sitt opphav i HTML INPUT elementer. Hvert tall i listen kan derfor være en tekststreng som må konverteres til heltall før utregning.

Parameter *operation* kan ha en av følgende verdier:

- 'sum': Metoden skal finne summen av heltallene i *numberList*.
- 'produkt': Metoden skal finne produktet av heltallene i *numberList*.
- 'min': Metoden skal finne det minste tallet i *numberList*.
- 'max': Metoden skal finne det største tallet i *numberList*.

Metoden *calculate* skal gjøre svaret fra utregningen tilgjengelig i egenskapen *result*, og egenskapen *status* er en tekststreng med informasjon om utregningen.

Egenskapen *status* kan ha en av følgende verdier:

- 'Ingen tall i tallisten'
  - Angir at *numberList* ikke har noen elementer som er heltall.
- 'Tallisten inneholder verdi(er) som ikke er tall'
  - Angir at *numberList* har noen elementer som ikke er heltall. Disse har blitt ignorert i utregningen.
- 'Alle input-verdier ble prosessert'
  - Alle elementer i *numberList* er heltall og har blitt brukt i utregningen.

Eksempelet under viser bruk av **Calculator**:

```
const calculator = new Calculator()
calculator.calculate('sum', ['1','3','7'])
console.log(`Svaret er: ${calculator.result}`)
console.log(`Status for utregning: ${calculator.status}`)
```

Under vises utskriften i nettleserkonsollet fra koden over:

Svaret er: 11

Status for utregning: Alle input-verdier ble prosessert

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

Løsningsforslag:

```
class Calculator {
  constructor() {
    /** @public {String} */ this.status = null
    /** @public {Number} */ this.result = null
  }
}
```

forts. neste side ...

```

/**
 * Metode for å utføre utregning
 * @public
 * @param {String} operation - Matematikk operasjon
 * @param {Array.<String>} numberList - Array av input-data
 *      som må konverteres til Array av heltall før utregning
 */
calculate(operation, numberList) {
    /**
     * Mulige operasjoner kunne vært definert i et static
     * objekt til klassen. Dette er ikke pensum i DAT108,
     * og foreløpig kun i et utkast til standard.
     */

    /** Konverterer til heltall */
    const numbers = numberList.map(value => value.trim()).filter(
        value => /^\\d+$/.test(value)).map(
        number => parseInt(number,10))

    if (numbers.length == 0) {
        this.status = 'Ingen tall i tallisten'
    } else {
        if (numbers.length == numberList.length) {
            this.status='Alle input-verdier ble prosessert'
        } else {
            this.status='Tallisten inneholder verdi(er) som ikke er tall'
        }

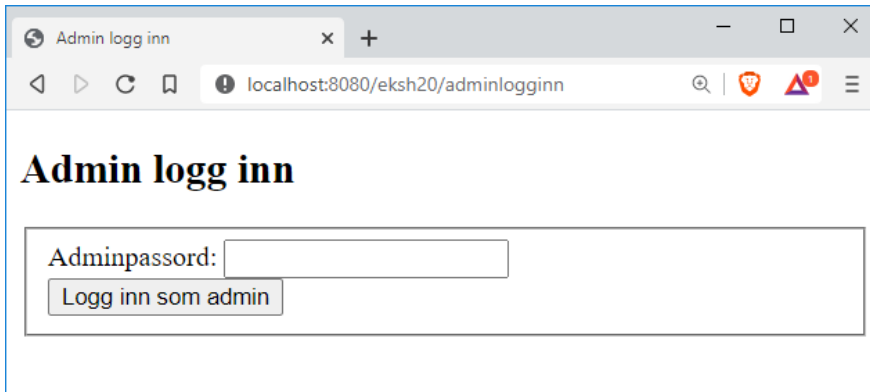
        switch (operation) {
            case 'sum':
                this.result = numbers.reduce(
                    (sum, number) => sum + parseInt(number)
                )
                break
            case 'produkt':
                this.result = numbers.reduce(
                    (product, number) => product * parseInt(number)
                )
                break
            case 'min':
                this.result = Math.min(...numbers)
                break
            case 'max':
                this.result = Math.max(...numbers)
                break
            default:
                this.status = 'Ukjent operasjon'
        }
    }
}
}

```

## Oppgave 4 (40% ~ 96 minutter) – Webapplikasjoner, tjenerside

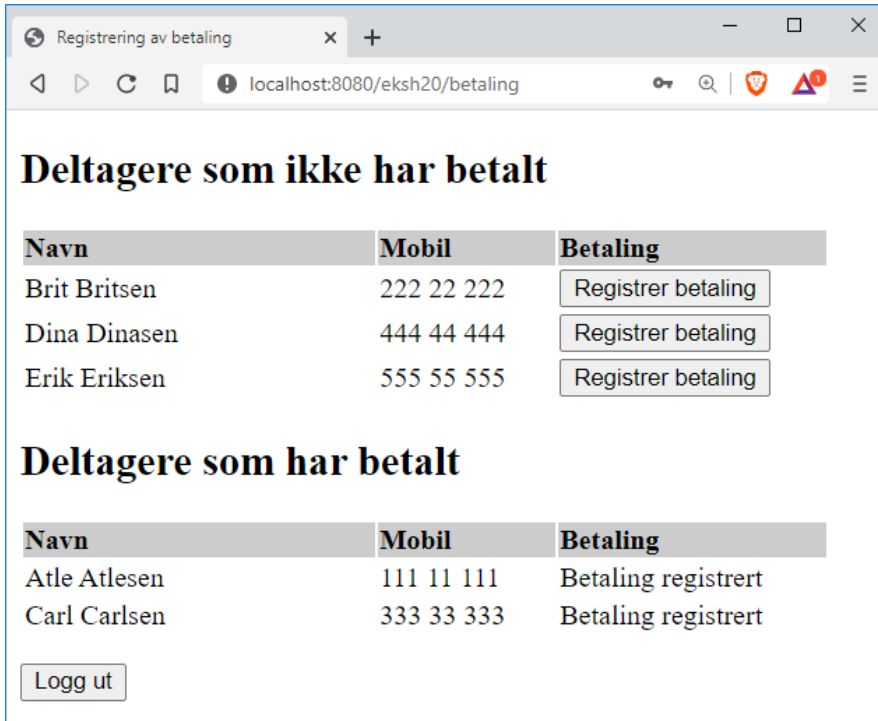
Vi skal se på en webapplikasjon der vi har en liste av deltagere påmeldt et arrangement. Arrangementet koster penger, og vi vil lage en løsning for å registrere og holde oversikt over innkomne betalinger.

Man må være innlogget som en slags "admin" for å kunne se betalingsoversikt og for å kunne registrere betalinger. Man logger seg inn som "admin" via et hardkodet hemmelig passord:



figur 4.1

Etter vellykket admin-innlogging kommer man til betalingsoversikten, som ser slik ut:



Navn	Mobil	Betaling
Brit Britsen	222 22 222	Registrer betaling
Dina Dinassen	444 44 444	Registrer betaling
Erik Eriksen	555 55 555	Registrer betaling

Navn	Mobil	Betaling
Atle Atlesen	111 11 111	Betaling registrert
Carl Carlsen	333 33 333	Betaling registrert

figur 4.2

Når man trykker på knappen "Registrer betaling" blir betaling registrert i databasen for aktuell deltager og man får opp betalingsoversikten på nytt.

F.eks. hvis man trykker på knappen for Dina Dinassen i figur 4.2 vil resultatsiden se slik ut:

Registrering av betaling

localhost:8080/eksh20/betaling

### Deltagere som ikke har betalt

Navn	Mobil	Betaling
Brit Britsen	222 22 222	Registrer betaling
Erik Eriksen	555 55 555	Registrer betaling

### Deltagere som har betalt

Navn	Mobil	Betaling
Atle Atlesen	111 11 111	Betaling registrert
Carl Carlsen	333 33 333	Betaling registrert
Dina Dinassen	444 44 444	Betaling registrert

Logg ut

figur 4.3

Hvis man prøver å se betalingsoversikten eller prøver å registrere betaling uten å være innlogget som admin skal man omdirigeres til admin-logginn med melding om at innlogging er påkrevd, slik:

Admin logg inn

Forespørselen din krever pålogging som admin

Adminpassord:

Logg inn som admin

figur 4.4

Utlogging fra betalingsoversikten ("Logg ut"-knappen) eller feil adminpassord ved pålogging skal også gi en omdirigering til admin logginn-siden.

Java-klasser og JSP-sider som skal være med i løsningen:

**/adminlogginn** mappes til **AdminLogginnServlet.java**

- `init()` har ansvar for å hente inn "adminpassord" fra `web.xml` (en `init-parameter`) og lagre det i en instansvariabel for senere oppslag.
- `doGet()` har ansvar for å få opp logginn-skjema med evt. feilmeldinger
  - ... i samarbeid med **adminlogginn.jsp**
- `doPost()` har ansvar for å logge brukeren inn som admin

**/betaling** mappes til **BetalingServlet.java**

- `doGet()` har ansvar for å få opp betalingsoversikten
  - ... i samarbeid med **betaling.jsp**
- `doPost()` har ansvar for å få registrert en betaling

**/adminloggut** mappes til **AdminLoggutServlet.java**

- `doPost()` har ansvar for å logge ut admin-brukeren

**@Entity** public class **Deltager** //JPA-entitet for deltagerne som er lagret i databasen

```
@Id private String mobil
private String navn
private boolean betalt
... gettere og settere
```

**DeltagerDAO.java** //Hjelpeklasse for databaseoperasjoner

- `List<Deltager> finnAlle()` //Henter ut en liste over alle registrerte deltagere
- `void registrerBetalingFor(mobil)` //Registrerer betaling for deltager med gitt mobil
- ...

**InnloggingUtil.java** //Hjelpeklasse for adgangskontroll, innlogging og utlogging

- `static void adminLogginn(request)` //Logger bruker i denne sesjonen inn som admin
- `static void adminLoggut(request)` //Logger bruker i denne sesjonen ut som admin
- `static boolean isInnloggetSomAdmin(request)` //Om bruker er innlogget som admin
- ...

Oppgaven deres er å implementere deler av denne løsningen. Det dere ikke skal implementere selv kan dere anta finnes og virker.

## Krav til løsningen

For å få full score må du løse oppgaven på best mulig måte i hht. prinsippene i kurset, dvs. Model-View-Controller, Post-Redirect-Get, EL og JSTL i JSP-ene, trådsikkerhet, robusthet, unntakshåndtering, god bruk av hjelpeklassene, elegant kode, osv ...

Dere trenger ikke ha med import-setninger eller direktiver i løsningene deres.

## Opgavene

a) Skriv hele **AdminLogginnServlet.java**

**Merknad til hele Oppg4:**

Dette er en påbygning på Oblig4, og det er naturlig at besvarelsene inneholder mye klipp og lim fra denne. Derfor er det "de nye tingene" fra eksamensoppgaven som må tillegges størst vekt.

Løsningsforslag (10% - 24 minutter):

```
@WebServlet("/adminlogginn")
public class AdminLogginnServlet extends HttpServlet {

    private String adminpassord;

    @Override
    public void init() ... {
        adminpassord = getServletConfig().getInitParameter("adminpassord");
    }

    @Override
    protected void doGet(...) ... {
        String errorMessage = "";

        //Antar at vi kan redirectes hit med 'flagg' for ulike feilsituasjoner
        if (request.getParameter("requiresLogin") != null) {
            errorMessage = "Forespørselen din krever pålogging som admin";
        } else if (request.getParameter("invalidPassword") != null) {
            errorMessage = "Ugyldig adminpassord";
        }
        request.setAttribute("redirectErrorMessage", errorMessage);
        request.getRequestDispatcher(
            "WEB-INF/adminlogginn.jsp").forward(request, response);
    }

    @Override
    protected void doPost(...) ... {
        String password = request.getParameter("password");
        TIPS! Bruk equals()!!!!. Ikke skriv "password != adminpassord" el.l.
        if (password == null || !password.equals(adminpassord)) {
            response.sendRedirect("adminlogginn?invalidPassword");
        } else {
            InnloggingUtil.adminLogginn(request);
            response.sendRedirect("betaling");
        }
    }
}
```

b) Skriv **doGet()** i **BetalingServlet.java**

Løsningsforslag (5% - 12 minutter):

```
@Override
protected void doGet(...) ... {

    if (!InnloggingUtil.isLoggetInnSomAdmin(request)) {
        response.sendRedirect("adminlogginn?requiresLogin");
    } else {
        List<Deltager> deltagere = deltagersDAO.finnAlle();

        request.setAttribute("deltagere", deltagere);

        request.getRequestDispatcher(
            "WEB-INF/betaling.jsp").forward(request, response);
    }
}
```

Alternativt kan man dele opp listen i to lister allerede her:

```
List<Deltager> deSomHarBetalt = deltagere.stream()
    filter(d -> d.isBetalt()).collect(Collectors.toList());
List<Deltager> deSomIkkeHarBetalt = deltagere.stream()
    filter(d -> !d.isBetalt()).collect(Collectors.toList());

request.setAttribute("deSomHarBetalt", deSomHarBetalt);
request.setAttribute("deSomIkkeHarBetalt", deSomIkkeHarBetalt);
```

I så fall trenger vi ikke `<c:if>` i JSP. ...



c) Skriv **Betaling.jsp**

**Tips:** Siden inneholder mange "Registrer betaling"-knapper. En grei måte å skille disse fra hverandre er å ha én <form>-tag per knapp.

Her er det sannsynligvis mye klipp og lim fra Oblig4. Uvedkommende ting trekkes ned, og det som er unikt for eksamensoppgaven vektlegges mest.

Løsningsforslag (10% - 24 minutter). Viktige ting uthevet:

```
<!DOCTYPE html>
<html>
<body>
<h2>Deltagere som ikke har betalt</h2>
<table><tr>
    <th>Navn</th>
    <th>Mobil</th>
    <th>Betaling</th>
</tr>
<c:forEach var="d" items="${deltagere}">
    <c:if test="${!d.betalt}"> <!-- Test ikke nødv. hvis to lister --%>
    <tr>
        <td>${d.navn}</td>
        <td>${d.mobil}</td>
        <td><form action="betaling" method="post">
            <input type="hidden" name="mobil" value="${d.mobil}" />
            <input type="submit" value="Registrer betaling" />
        </form></td> <!-- Slipper hidden hvis man bruker --%>
    </tr>
    </c:if>
</c:forEach>
</table>

<h2>Deltagere som har betalt</h2>
<table><tr bgcolor="#cccccc">
    <th>Navn</th>
    <th>Mobil</th>
    <th>Betaling</th>
</tr>
<c:forEach var="d" items="${deltagere}">
    <c:if test="${d.betalt}"> <!-- Test ikke nødv. hvis to lister -->
    <tr>
        <td>${d.navn}</td>
        <td>${d.mobil}</td>
        <td>Betaling registrert</td>
    </tr>
    </c:if>
</c:forEach>
</table>
<p>
    <form action="adminloggut" method="post">
        <input type="submit" value="Logg ut" />
    </form>
</p>
</body>
</html>
```

d) Skriv **doPost()** i **BetalingServlet.java**

Løsningsforslag (5% - 12 minutter):

```
@Override
protected void doPost(...) ... {

    if (InnloggingUtil.isLoggetInnSomAdmin(request)) {

        // Parameterverdien fra hidden-/button-input fra JSP.
        // Kunne kanskje også validert denne, men feil verdi her
        // skyldes hacking. Du er innlogget som admin, og den er
        // tilsendt via trykk på knapp.
        String mobil = request.getParameter("mobil");

        deltagerDAO.registrerBetalingFor(mobil);

        response.sendRedirect("betaling");

    } else {
        response.sendRedirect("adminlogginn?requiresLogin");
    }
}
```

e) Skriv **adminLogginn(request)** i **InnloggingUtil.java**

Løsningsforslag (5% - 12 minutter):

```
public static void adminLogginn(HttpServletRequest request) {  
  
    // Invalidate old session  
    HttpSession session = request.getSession(false);  
    if (session != null) {  
        session.invalidate();  
    }  
  
    // Alternativt, litt mer "quick and dirty":  
    request.getSession(true).invalidate();  
  
    // Eller kanskje aller best, siden denne er listet i API-et:  
    adminLoggut(request);  
  
    // Create new fresh session  
    HttpSession sesjon = request.getSession(true);  
  
    // Important! Store some arbitrary token in the session  
    // as an identifier of the logged-in-session. Having an  
    // active session is in itself not enough "proof" that you  
    // are logged in. Vet ikke hvorfor jeg skrev dette på engelsk.  
    sesjon.setAttribute("loggedInAs", "admin");  
}
```

f) Skriv **registrerBetalingFor(mobil)** i **DeltagerDAO.java**

Løsningsforslag (5% - 12 minutter):

```
public void registrerBetalingFor(String mobil) {  
  
    Deltager deltager = em.find(Deltager.class, mobil);  
    if (deltager != null) {  
        deltager.setBetalt(true);  
    }  
}
```

Merknad: Det er ikke nødvendig med `em.merge(deltager)` siden den allerede er i managed tilstand.

Merknad: Hvis man ikke gjør null-sjekk (som man burde gjøre) er registreringen en one-liner: `em.find(Deltager.class, mobil).setBetalt(true);`

## Oppgave 5 (16% ~ 38 minutter) – Tråder

- a) Vi ønsker å legge inn en tråd i et program som gjør "bitcoin-mining" i bakgrunnen.

"Bitcoin-mining" i denne sammenhengen betyr helt enkelt å gå gjennom en for-løkke for alle positive (long) heltall, beregne en hash for hvert enkelt tall, og sjekke om verdien av hashen starter med fem nullere, altså `.startsWith("00000")`. I så fall skal tallet og hashen skrives ut på skjermen. F.eks. slik:

`262997966 -> 00000PM477MtJDRsn+9RjMhPooy/12F4Vj7a+WbbZ30=`

Du kan anta at du har en metode `static String hashOf(long number)` som utfører selve hashingen av tallet.

Tråden skal fortsette beregningene til alle tall t.o.m. `Long.MAX_VALUE` er gjennomløpt.

Skriv en **main-metode** som oppretter og starter en tråd som gjør slik "bitcoin-mining". (Vi får innbille oss at main også gjør andre ting etterpå, men det tenker vi ikke på her). TIPS: Lag en **hjelpemetode** for selve logikken som skal utføres i tråden slik at `run()` kun trenger å kalle denne metoden.

Løsningsforslag:

Hjelpemetoden:

```
public static void mineBitcoin() {
    for (long i = 0; i <= Long.MAX_VALUE; i++) {
        if (hashOf(i).startsWith("00000")) {
            System.out.println(i + " -> " + hashOf(i));
        }
    }
}
```

Hjelpemetoden, alternativ:

```
public static void mineBitcoin() {
    LongStream.rangeClosed(0, Long.MAX_VALUE)
        .parallel() //evt. for å utnytte flere kjerner samtidig
        .filter(i -> hashOf(i).startsWith("00000"))
        .forEach(i -> System.out.println(i + " -> " + hashOf(i)));
}
```

Merknad: Innholdet i metoden `mineBitcoin()` er ikke det sentrale her. Oppgaven dreier seg om å opprette og starte en tråd i `main()` som kaller en gitt metode fra `run()`. Det verken gis mye poeng for, eller trekkes mye for, hvor bra selve "miningen" (over) er gjort.

... se neste side for resten ...

Løsningsforslag (hoveddel):

```
public static void main(String[] args) {
    new Thread(() -> mineBitcoin()).start();
}
```

Fem alternative løsninger:

```
1 -----
    public static void main(String[] args) {
        Runnable r = () -> mineBitcoin();
        Thread t = new Thread(r);
        t.start();
    }
2 -----
    public class BitcoinRunnable implements Runnable {
        @Override
        public void run() {
            mineBitcoin();
        }
    }

    public static void main(String[] args) {
        Thread bitcoinThread = new Thread(new BitcoinRunnable());
        bitcoinThread.start();
    }
3 -----
    public class BitcoinThread extends Thread {
        @Override
        public void run() {
            mineBitcoin();
        }
    }

    public static void main(String[] args) {
        Thread bitcoinThread = new BitcoinThread();
        bitcoinThread.start();
    }
4 -----
    public static void main(String[] args) {
        Thread bitcoinThread = new Thread(new Runnable() {
            @Override
            public void run() {
                mineBitcoin();
            }
        });
        bitcoinThread.start();
    }
5 -----
    public static void main(String[] args) {
        Thread bitcoinThread = new Thread() {
            @Override
            public void run() {
                mineBitcoin();
            }
        };
        bitcoinThread.start();
    }
```

- b) Vi skal lage et lite program der flere tråder disponerer en felles "bankkonto". Bankkontoen skal blokkere mot overtrekk, dvs. at tråder som ønsker å gjøre uttak som vil føre til negativ saldo må vente til det er satt inn nok penger før uttaket blir gjennomført.

Skriv en **main-metode** der det opprettes et Bankkonto-objekt og to tråder som gjør en del innskudd og uttak på denne bankkontoen. Skriv også **Bankkonto**-klassen. (PS! Du trenger ikke å ha med kode for unntakshåndtering.)

Generell merknad til oppgaven:

Det står at "**Bankkontoen skal blokkere** mot overtrekk, dvs. at tråder som ønsker å gjøre uttak som vil føre til negativ saldo må vente til det er satt inn nok penger før uttaket blir gjennomført."

Dette må tolkes som at det er Bankkonto-klassen som skal ha denne funksjonaliteten, ikke evt. tråder som bruker bankkontoen. Altså: **synchronized, wait og notify** (evt. lock, unlock, await og signal) må være i Bankkonto. Det er dette som er det sentrale i oppgaven.

Løsningsforslag (har fjernet/utelatt unntakshåndtering):

```
public class Bankkonto {  
  
    int saldo;  
  
    public Bankkonto(int saldo) {  
        this.saldo = saldo;  
    }  
  
    public synchronized void taUt(int uttaksbelop) {  
        while (saldo < uttaksbelop) {  
            wait();  
        }  
        saldo -= uttaksbelop;  
    }  
  
    public synchronized void settInn(int innskuddsbelop) {  
        saldo += innskuddsbelop;  
        notifyAll();  
    }  
}
```

Alternativ løsning neste side ...

Alternativ løsning med Lock, Condition, await() og signal():

```
public class Bankkonto {

    int saldo;

    private Lock laasen = new ReentrantLock();
    private Condition pengerSattInn = laasen.newCondition();

    public Bankkonto(int saldo) {
        this.saldo = saldo;
    }

    public void taUt(int uttaksbelop) {
        laasen.lock();
        try {
            while (saldo < uttaksbelop) {
                pengerSattInn.await();
            }
            saldo -= uttaksbelop;
        } finally {
            laasen.unlock();
        }
    }

    public void settInn(int innskuddsbelop) {
        laasen.lock();
        try {
            saldo += innskuddsbelop;
            pengerSattInn.signalAll();
        } finally {
            laasen.unlock();
        }
    }
}

public static void main(String[] args) {

    Bankkonto felleskontoen = new Bankkonto(2000);

    Thread anne = new Thread() {
        @Override public void run() {
            felleskontoen.taUt(2000);
            felleskontoen.taUt(2000);
        }
    };

    Thread knut = new Thread() {
        @Override public void run() {
            sleep(3000); felleskontoen.settInn(1000);
            sleep(3000); felleskontoen.settInn(1000);
        }
    };

    anne.start();
    knut.start();
}
```