Høgskulen på Vestlandet

~~EKSAMENSOPPGAVE~~

Løsningsforslag
m/sensurnotater

Emnekode: DAT108

Emnenavn: Programmering og webapplikasjoner

Klasse: 2. klasse DATA / INF

Dato: 9. desember 2021

Eksamensform: Skriftlig hjemmeeksamen (Wiseflow | FLOWassign)

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

Antall eksamensoppgaver: 5

Antall sider (medregnet denne): 16

Antall vedlegg: Ingen

Tillatte hjelpemiddel: Alle.

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

Generelle merknader som gjelder hele eksamen:

Dette er en hjemmeeksamen med alle tilgjenglige hjelpemidler. Det er da naturlig at spørsmålene fokuserer litt mer på forståelse enn på fakta. Slike spørsmål oppfattes ofte som **vanskeligere**.

Det viser seg at mange studenter velger å lage kjørbare programmer på eksamen, og bruker det som trygghet for at svarene er brukbare. Dette tar nok mer tid enn kun å notere ned svaret som "tekst", og det kan kanskje være én grunn til at det er lett å komme i **tidsnød**. En del besvarelser inneholder fulle implementasjoner av klasser som det ikke spørres etter på eksamen. Å gjennomføre en programmerings-eksamen med digitalt "blyant og papir" er kanskje noe man burde øvd på.

Vi sliter med (antatt) omfattende juks med denne eksamensformen, altså at studenter snakker sammen og kopierer løsninger av hverandre, eller deler løsninger de har funnet på nett. Ved å stille litt mer åpne spørsmål, og gi litt mer valgfrihet i løsninger, vil det være mer unaturlig og avslørende å ha identiske svar. Men spørsmål med **rom for variasjon** kan nok være litt krevende.

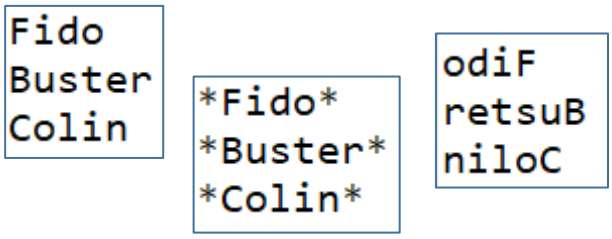
Oppgave 1 (20% ~ 48 minutter) – Lambda-uttrykk og strømmer

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

a) Tenk at du har en liste av hundenavn slik:

```
List<String> hunder = List.of("Fido", "Buster", "Colin");
```

Ved å bruke **forEach(c)**-metoden på listen med ulike parametere **c** kan du lage 3 ulike utskrifter, slik:



Skriv de 3 ulike variantene av parameteren **c** og lagre dem i variablene **printPlain**, **printMedStjerner** og **printBaklengs**. Husk å få med datatypen til variablene. *Tips til baklengs: new StringBuilder(s).reverse() vil snu strengen s.*

Løsningsforslag:

```
//Unødvendig hjelpemetode, men den gjør printBaklengs mer leselig:
String baklengs(String s) {
      return new StringBuilder(s).reverse().toString();
}

Consumer<String> printPlain       = System.out::println;
Consumer<String> printMedStjerner = s -> System.out.println("*"+s+"*");
Consumer<String> printBaklengs    = s -> System.out.println(baklengs(s));
```

Sensurnotater:

Selv om dette var tenkt som en enkel oppgave er det en del som har bommet / misforstått, f.eks. brukt Function som mapper fra String til String.

Jeg synes selv at oppgaven er klar. Det står at den gjelder "forEach(c)-metoden på listen" og at man skal lage de "3 ulike variantene av parameteren c" som gir de 3 ulike utskriftene.

forEach()-metoden er definert som **forEach(Consumer<? super T> action)**. Parameteren er altså av typen **Consumer**. I kurset har vi sett på mange eksempler med både forEach() og med Consumer.

I de neste deloppgavene skal vi se på en liste av bøker. En Bok har disse egenskapene:

```
class Bok {
   public String tittel;
   public int aar;
   public List<String> forfattere; //Hver forfatter er en enkel String

   ... konstruktører og metoder ...
}
```

Tenk så at vi f.eks. har en liste med bøker slik:

```
List<Bok> boker = List.of(
      new Bok("Core Java Volume I", 2022, List.of("Cay Horstmann")),
      new Bok("Effective Java", 2017, List.of("Cay Horstmann")),
      ...
      new Bok("Head First Java", 2005,
            List.of("Kathy Sierra", "Bert Bates")),
      new Bok("Java Concurrency in Practice", 2006,
            List.of("Brian Goetz", "Tim Peierls", "Joshua Bloch"))
      );
```

Du skal nå løse noen oppgaver ved å bruke streams og lambda-uttrykk.

b) Skriv en setning som lager en ny liste av alle bøker som er utgitt i 2015 eller tidligere

```
Løsningsforslag:

List<Bok> for2015 = boker.stream()
        .filter(b -> b.aar <= 2015)
        .collect(Collectors.toList());
```

c) Skriv en setning som lager en liste av alle boktitler som inneholder ordet Java

```
Løsningsforslag:

List<String> titlerMedJava = boker.stream()
      .filter(b -> b.tittel.contains("Java")) //evt. kan denne gjøres etter map
      .map(b -> b.tittel)
      .collect(Collectors.toList());
```

```
Sensurnotater:

Denne oppgaven ser det ut som om mange har feiltolket.

Mens det i b) står "en liste av bøker", står det her i c) "en liste av boktitler".
Poenget med oppgaven er altså å dra tittelen på boken ut av bok-objektene. Det
gjøres med map().

Besvarelsene tyder på at oppgaven lett kunne misforstås, og det er derfor gitt
nesten full score selv om man kun har gjort filter(), men ikke map().
```

d) Skriv en setning som lager en liste av alle forfatterne uten duplikater

Løsningsforslag:

```
List<String> forfattere = boker.stream()
      .flatMap(b -> b.forfattere.stream())
      .distinct()
      .collect(Collectors.toList());
```

Sensurnotater:

Her tenkte jeg at vi kunne se på en litt mindre vanlig operasjon enn filter, map, forEach og collect. Nemlig flatMap!

Eksemplet er nesten likt det vi så på med utviklere som kunne flere språk, og vi ønsket en liste av språkene uten duplikater. Minner igjen om at dette er hjemmeeksamen med alle hjelpemidler. Hvis man gjenkjenner at man har en liste av objekter, og hvert objekt har igjen en liste, så kunne man funnet løsningen i eksemplet gjennomgått i undervisningen.

e) Utvid d) med at forfatterne sorteres på etternavn ved å bruke Stream<T> sin sorted(Comparator<? super T> comparator) der comparatoren er tilordnet en variabel kalt **paaEtternavn** før bruk. Tips: s.substring(s.lastIndexOf(" ")) gir etternavnet for navnet s.

Løsningsforslag:

```
//Unødvendig hjelpemetode, men den gjør svaret mer leselig:
String etternavn(String s) {
      return s.substring(s.lastIndexOf(" ");
}

Comparator<String> paaEtternavn =
        (s1, s2) -> etternavn(s1).compareTo(etternavn(s2));
... alternativt
Comparator<String> paaEtternavn = Comparator.comparing(s -> etternavn(s)));
... eller (der Oppg1e er klassen hjelpemetoden er definert i)
Comparator<String> paaEtternavn = Comparator.comparing(Oppg1e::etternavn));

List<String> forfattere = boker.stream()
      .flatMap(b -> b.forfattere.stream())
      .distinct()
      .sorted(paaEtternavn)
      .collect(Collectors.toList());

... alternativt, ved å bruke resultatet fra d)
forfattere = forfattere.stream()
      .sorted(paaEtternavn)
      .collect(Collectors.toList());
```

Sensurnotater:

Dette siste delspørsmålet var ment å være litt vanskelig, og det var ikke beregningen at flertallet ville klare å løse denne. Vi har riktignok laget mange comparatorer på forelesninger og i øvinger, og det burde være velkjent.

Oppgave 2 (20% ~ 48 minutter) – JavaScript

a) HTML taggen *script* kan brukes sammen med et attributt *defer*.

i. Hva er konsekvensen av *defer* attributtet. Kan *defer* ha noe å si for ytelsen til webapplikasjonen? Svaret må begrunnes.

Attributtet *defer* forteller nettleser at koden kan lastes parallelt med resten at web-dokumentet, men koden blir først kjørt etter at selve web-dokumentet er ferdig analysert.

Dette kan ha en konsekvens for ytelsen da nettleser kan laste koden i en egen I/O tråd parallelt med å hente og analysere resten av web-dokumentet.

ii. Hvor i et webdokument bør *script* tagg med *defer* plassers? Svaret må begrunnes.

Den bør plasseres i head-delen. Da kan HTML-koden nedenfor i dokumentet lastes og analyseres parallelt med å laste JavaScript-koden. All kode i web-dokumentet foran script-taggen vil bli lastet og analysert før JavaScript-koden blir lastet. Derfor er det bedre jo tidligere i web-dokumentet denne taggen plasseres.

iii. Hvor i et webdokument bør *script* tagg uten *defer* plasseres? Svaret må begrunnes.

Den bør plasseres helt i slutten av dokumentet, i slutten av body- eller html-delen.

Når web-dokument blir lastet og nettleser møter script taggen vil JavaScript-koden bli kjørt der og da. Dette har flere konsekvenser.

JavaScript-kode kan kun aksessere DOM-strukturer som allerede er analysert av nettleser. Derfor vil all HTML-koden nedenfor script-taggen være utilgjengelig for JavaScript-koden.

JavaScript-koden blir kjørt av samme tråd som bygger DOM-strukturen. HTML nedenfor script-taggen blir derfor ikke vist i nettleser før JavaScript-koden er ferdig med å kjøre. Dette kan gi dårlig brukeropplevelse hvis JavaScript-koden er treg eller henger.

b) For å modifisere DOM-strukturen til et webdokument har vi blant annet egenskapene *innerHTML*, *textContent* og *innerText*, og metoden *insertAjacentHTML*.

i. Gi eksempler på når det er riktig å bruke *innerHTML* og *insertAjacentHTML*. Svaret må begrunnes.

innerHTML og *insertAdjacentHTML* kan brukes for å bygge rene HTML-skjeletter. HTML-koden må ikke inneholde data eller parameter som har sitt opphav utenfor metoden som oppretter skjelettet.

Eksempel på *constructor* i en klasse **MemberGUI**:

```
constructor() {
    this.#memberTable = document.createElement("table");

    const tableContent = `
    <thead>
        <tr><th>Fornavn</th><th>Etternavn</th></tr>
    </thead>
    <tbody>
    </tbody>`;

    this.#memberTable.insertAdjacentHTML("beforeend", tableContent);
}
```

ii. Gi eksempler på situasjoner der *innerHTML* og *insertAjacentHTML* ikke må brukes. Svaret må begrunnes.

innerHTML og *insertAdjacentHTML* må ikke brukes for å legge inn data på websiden som har opphav utenfor metoden med *innerHTML* eller *insertAdjacentHTML*.

Metoden nedenfor er en metode i klassen **MemberGUI**, og viser feil bruk av *insertAdjacentHTML*.

```
addMemberBad(firstname, lastname) {
    const tableRowString = `
        <tr><td>${firstname}</td><td>${lastname}</td></tr>
    `;

    const tbodyElement = this.#memberTable.tBodies[0];
    tbodyElement.insertAdjacentHTML("beforeend", tableRowString);
}
```

iii. Gi eksempler på når det er riktig å bruke *textContent* og *innerText*. Svaret må begrunnes.

textContent og *innerText* brukes begge for å sette inn ren tekst i web-dokumentet. Eventuell HTML-kode vil bli omkodet og bli vist som ren tekst på websiden.

forts. neste side ...

Disse metodene må alltid brukes for å sette inn data på websiden som har opphav utenfor metoden med *textContent* eller *innerText*. En angriper som forsøker å ta kontroll over websiden ved f.eks. å fylle kode inn i et formskjema kan lykkes hvis data blir vist med *innerHTML* eller *insertAdjacentHTML*. Derimot vil koden bli ufarliggjort av *textContent* og *innerText*.

Metoden nedenfor er en metode i klassen **MemberGUI**, og viser riktig bruk av *insertAdjacentHTML* og *textContent*. HTML-skjelettet lages av *insertAdjacentHTML*, mens data blir fylt inn ved å bruke *textContent*.

```
addMemberGood(firstname, lastname) {
    const tableRowString = `<tr><td></td><td></td></tr>`;

    const tbodyElement = this.#memberTable.tBodies[0];
    tbodyElement.insertAdjacentHTML("beforeend", tableRowString);

    const lastRow = tbodyElement.lastElementChild;
    lastRow.cells[0].textContent = firstname;
    lastRow.cells[1].textContent = lastname;
}
```

c) Opprett en JavaScript klasse **Parking**. Instanser av klassen skal kunne administrere et parkeringsområde og parkeringsavgift for biler på området.

Et parkeringsområde har et gitt antall plasser for biler, og hver bil må tilhøre en takstgruppe. Klassen har metodene *addCar* og *removeCar* for å registrere at biler kjører inn og ut av parkeringsområdet.

Kodeeksempelet under demonstrer hvordan en instans av **Parking** kan bli opprettet:

```
const carpark= new Parking(50, { electric: 5, normal: 30 });
```

Første argument til konstruktøren er antall parkeringsplasser på parkeringsområdet, og andre argument er et objekt med takstgrupper og parkeringsavgift per påbegynt 15 minutt intervall. I eksempelet koster det 5 kroner per påbegynt 15 minutt intervall for biler i takstgruppe *electric* og 30 for biler i takstgruppe *normal*.

For biler som forlater området innen 10 minutter beregnes det ingen parkeringsavgift.

Klassen **Parking** har følgende metoder:

- *addCar(regno,taxgroup)*: Metoden registrerer at bil med registreringsnummer *regno* som er i takstgruppe *taxgroup* har kjørt inn på parkeringsområdet.

  Parametere:
  - Inn-parametere: **String**, **String**
  - Returverdi: **Object**

  Metoden returnerer et objekt, eller *null*. Returverdi *null* betyr at registrering av bil feilet. Grunner for returverdi *null* kan være at parkeringsområdet er fullt, eller at takstgruppen *taxgroup* ikke finnes.

Kodeeksempelet under registrerer at bilen «EK12345» som har takstgruppe «electric» har kjørt inn på parkeringsområdet:

```
const arrival = carpark.addCar("EK12345", "electric");
```

Returverdi er følgende objekt:

```
{
     regno: "EK12345",
     taxgroup: "electric",
     arrival: 1637065716428
}
```

Egenskapene *regno* og *taxgroup* viser parameterverdiene som ble brukt i metodekallet. Egenskapen *arrival* er tidspunktet for når bilen ble registrert inn, representert ved antall millisekunder siden 1. januar 1970.

- *removeCar(regno)*: Metoden brukes når bil med registreringsnummer *regno* forlater parkeringsområdet.

  Parametere:
  - Inn-parameter: **String**
  - Returverdi: **Object**

  Metoden returnerer et objekt, eller *null*. Returverdi er *null* hvis bilen ikke var registrert inn på parkeringsområdet.

  Kodeeksempelet under fjerner bilen «EK12345» fra *carpark*:

```
const departure = carpark.removeCar("EK12345");
```

  Returverdi er følgende objekt:

```
{
     regno: "EK12345",
     taxgroup: "electric",
     arrival: 1637065716428,
     departure: 1637070036428,
     cost: 25
}
```

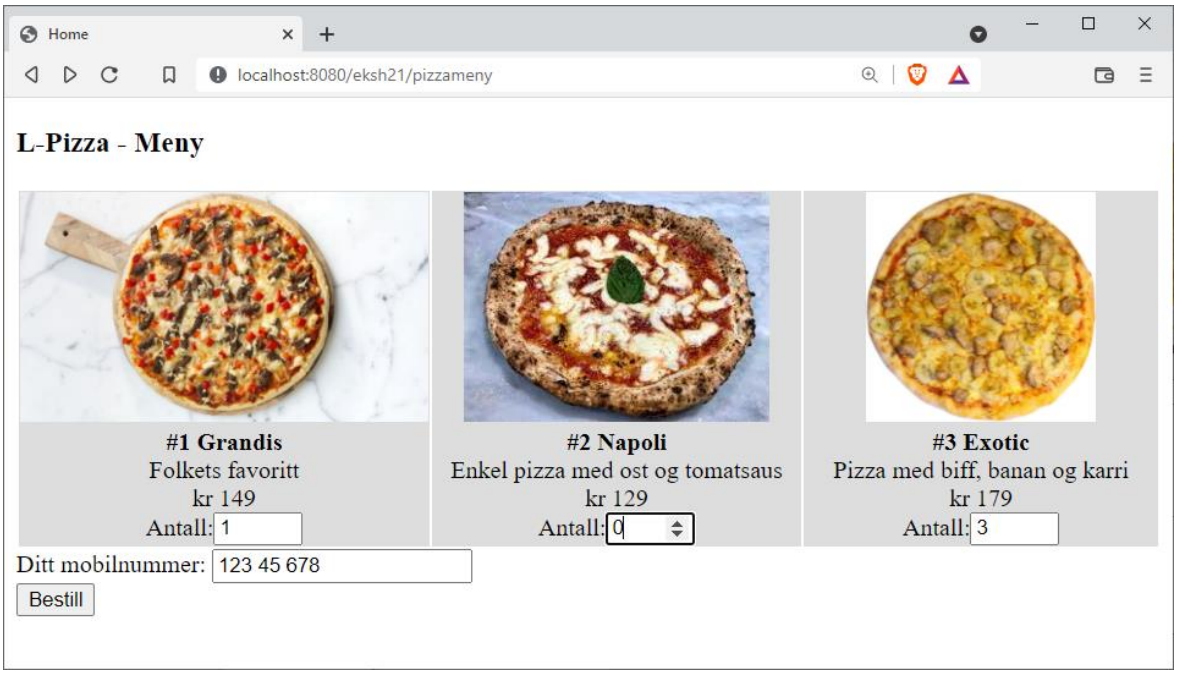
  Egenskapene *regno*, *taxgroup* og *arrival* er som for *addCar*. Egenskapen *departure* er tidspunktet for når bilen forlot parkeringsområdet, representert ved antall millisekunder siden 1. januar 1970. Egenskapen *cost* er parkeringsavgiften for denne parkeringen.

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

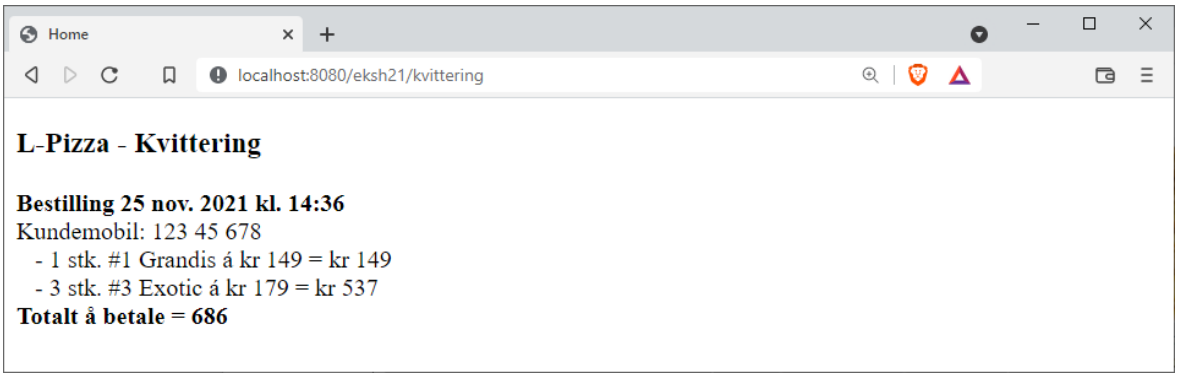
```
class Parking {
    #countTotal;
    #cars = new Map();
    #priceInfo;

    constructor(count, priceInfo) {
        this.#countTotal = count;
        this.#priceInfo = priceInfo;
    };

    get isFull() { return this.#cars.size >= this.#countTotal }

    addCar(regno, taxgroup) {
        if (regno === undefined) return null;
        if (taxgroup === undefined) return null;
        if (this.#cars.has(regno)) return null;
        if (!this.#priceInfo.hasOwnProperty(taxgroup)) return null;
        if (this.isFull) return null;

        const arrival = Date.now();
        this.#cars.set(regno, { taxgroup: taxgroup, arrival: arrival });
        return { regno: regno, taxgroup: taxgroup, arrival: arrival };
    }

    removeCar(regno) {
        if (regno === undefined) return null;
        if (!this.#cars.has(regno)) return null;

        const { taxgroup, arrival } = this.#cars.get(regno);
        const fee = this.#priceInfo[taxgroup];
        const departure = Date.now();
        const cost = this.#calculateCost(fee, arrival, departure);

        this.#cars.delete(regno);
        return {
            regno: regno,
            taxgroup: taxgroup,
            arrival: arrival,
            departure: departure,
            cost: cost
        };
    }

    #calculateCost(fee, start, end) {
        const ms = end - start;
        const seconds = ms / 1000;
        const minutes = seconds / 60;

        let cost = 0;
        if (minutes > 10) {
            const intervals = Math.ceil(minutes / 15);
            cost = intervals * fee;
        }

        return cost;
    }
}
```

Oppgave 3 (40% ~ 96 minutter) – Webapplikasjoner, tjenerside

Dere skal lage litt av en løsning for online bestilling hos en pizza-restaurant.



Bilde 3.1 - /pizzameny



Bilde 3.2 - /kvittering

Vi begynner med en forklaring av hvordan applikasjonen er bygget opp.

Informasjonen om pizzaene i menyen er lagret i en databasetabell **pizza**,

nr [PK] integer	navn character varying	beskrivelse character varying	pris integer	bildefil character varying
1	Grandis	Folkets favoritt	149	grandis.jpg
2	Napoli	Enkel pizza med ost og tomatsaus	129	napoli.jfif
3	Exotic	Pizza med biff, banan og karri	179	exotic.jpg

Bilde 3.3 - Databasetabell med eksempeldata

Vi har tilsvarende JPA entitetsklasse i Java kalt **Pizza**.

Navn på bildefil refererer til et bilde som ligger i applikasjonen under **.../webapp/bilder/**

Vi har en hjelpeklasse (@Stateless EJB) **PizzaDAO** som vi kan bruke til å lese inn data fra databasen til applikasjonen. Metoden som kan brukes heter **List<Pizza> hentAllePizzaer()**.

Applikasjonen skal hente inn dataene ved/før første gangs bruk og deretter holde dem i minnet slik at de er tilgjengelig for alle deler av applikasjonen, dvs. alle Servleter og alle JSP-er, på en grei måte. *(Hint: Hvilket scope er egnet til dette?)*

Pizzameny-siden skal bygges opp på grunnlag av disse dataene. For enkelhets skyld kan vi si at siden viser en tabell av pizzaer med én rad og n kolonner. Se Bilde 3.1.

Bestilling gjøres ved at man velger antall av de ulike pizzaene, gir inn mobilnummer for referanse og trykker Bestill (~ **POST /pizzameny**).

Det skal da lages et **Bestilling**-objekt, dette skal lagres i databasen via **PizzaDAO** sin **void lagreBestilling(Bestilling b)**. Deretter skal kvitteringssiden vises som i Bilde 3.2.

Et **Bestilling**-objekt inneholder følgende data

- **tidspunkt for bestillingen** (LocalDateTime)
- **kundens mobil** (String)
- **antall av de ulike pizzaene** (f.eks. Map av <Pizza, Integer>). *Grunnen til at hele Pizzaen foreslås brukt som nøkkel er at man da enkelt kan få alle pizzaene ut ved å bruke metoden **Map.keySet()**. Det er kanskje/mer like ryddig å bruke pizza.nr som nøkkel. Velg den måten du foretrekker av disse.*

og konstruktører og metoder **etter behov**.

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**, **ufarliggjøring** av brukerinput, **unntakshåndtering**, **god bruk av hjelpeklassene**, **elegant kode**, osv ...

Løsningen trenger ikke å være robust mot tekniske databasefeil. Du kan anta at det hentes minst ett pizza-objekt fra databasen når du spør etter det.

Løsningen trenger ikke å være robust mot «hacking», dvs. utilsiktet bruk. Du kan anta at nødvendig validering er gjort i nettleseren via HTML eller JavaScript. Altså at antall som skal bestilles av de ulike pizzaene er 0 eller et positivt heltall av rimelig størrelse og at mobilnummer er noe som godtas av applikasjonen.

Oppgaver

a) (15%) Skriv **PizzamenyServlet** som mappes til URLen **/pizzameny**. Tilhørende view er pizzameny.jsp.

Løsningsforslag, del1 av a) - Teller 7%:

```
1 @WebServlet("/pizzameny")
public class PizzamenyServlet extends HttpServlet {

1     @EJB
1     private PizzaDAO pizzaDAO;

      @Override
1     public void init() {
2           List<Pizza> pizzaer = pizzaDAO.hentAllePizzaer();
2           getServletContext().setAttribute("pizzaer", pizzaer);
      }

      /** GET henter frem siden for pizzamenyen */
      @Override
      protected void doGet(...) {
2           request.getRequestDispatcher("WEB-INF/jsp/pizzameny.jsp")
                        .forward(request, response);
      }

      // forts. neste side ...
```

Sensurnotater:

Mange hadde klart denne greit, med unntak om at pizzaene kun skal hentes fra databasen én gang, dvs. ved oppstart. De fleste lagret også listen av pizzaer i request- eller session-scope, ikke i app-scope.

Ved å holde List<Pizza> i en instans-/objektvariabel (i tillegg til i app-scope) trenger vi ikke å hente den frem på nytt i doPost(). Det kan jo være en idé.

Løsningsforslag, del2 av a) - Teller 8%:

```
      /** POST tar imot en bestilling av et antall av de ulike pizzaene. */
      @Override
      protected void doPost(...) {

1           String kundemobil = request.getParameter("kundemobil");

            /* En bestilling inneholder tidspunkt, kundens mobil og antall av de
             * ulike pizzane. Tidspunkt og tomt "pizza-antall-map" settes internt
             * i konstruktøren, mens mobilnummeret er fra brukerinput.
             */
2           Bestilling bestilling = new Bestilling(kundemobil);

            /* Hvis ikke lagret i en instans-/objektvariabel i servleten. */
            List<Pizza> pizzaer =
                  (List<Pizza>) getServletContext().getAttribute("pizzaer");

            /* Nå er det å gå gjennom alle pizzaene i menyen, sjekke hvor mange
             * brukeren har bestilt av hver, og legge de til i bestillingen.
             * Ved å følge tipset i b) om at params kan hete "noe fast + pizza-nr"
             * vil parametrene f.eks. hete antallAv1, antallAv2, antallAv3, osv ...
             */
2           for (Pizza p : pizzaer) {
                  int antall = Integer.parseInt(
                          request.getParameter("antallAv" + p.getNr()));
                  if (antall > 0) {
                      bestilling.leggTil(p, antall);
                  }
            }

            //Alternativ til å bruke løkke:
            Function<Pizza, Integer> antallAv =
                p -> Integer.parseInt(request.getParameter("antallAv" + p.getNr()));
            pizzaer.stream()
                .filter(p -> antallAv.apply(p) > 0)
                .forEach(p -> bestilling.leggTil(p, antallAv.apply(p)));

2           pizzaDAO.lagreBestilling(bestilling);

2           request.getSession().setAttribute("bestilling", bestilling);

1           response.sendRedirect("kvittering");
      }
}
```

Sensurnotater:

Denne bør nok løses samtidig med (eller kanskje helst etter) 3b) pizzameny.jsp siden det er input derfra som skal behandles her. Etterpåklokskap: Jeg burde nok hatt doPost() som et eget delspørsmål etter pizzameny.jsp.

b) (10%) Skriv **pizzameny.jsp**. *Tips: Parameternavn for antall av de ulike pizzaene kan kanskje være en kombinasjon av noe fast + pizza-nr.*

Løsningsforslag:

Det vi prøver å generere er ca. noe slikt (dynamiske data merket med **fet blå**):

```
<html>
<body>
<h3>L-Pizza - Meny</h3>
<form method="post">
   <table><tr>
      <td><img src="bilder/grandis.jpg"><br>
            <b>#1 Grandis</b><br>
            Folkets favoritt<br>
            kr 149<br>
            Antall:<input type="text/number" name="antallAv1">
      </td>

            ... gjenta <td> for alle pizzaer nr 2, 3, 4, ... i databasen

   </tr></table>
   Ditt mobilnummer: <input type="text" name="kundemobil"><br>
   <input type="submit" value="Bestill">
</form>
</body>
</html>
```

--- Løsning. Listen av pizzaer kan nås via attributten "pizzaer" fra 3a):

```
<html>
<body>
<h3>L-Pizza - Meny</h3>
2<form method="post">
   <table><tr>
2 1    <c:forEach var="pizza" items="${pizzaer}">
1            <td><img src="bilder/${pizza.bildefil}" ><br>
1                  <b>#${pizza.nr} ${pizza.navn}</b><br>
                   ${pizza.beskrivelse}<br>
                   kr ${pizza.pris}<br>
2                  Antall:<input type="text/number" name="antallAv${pizza.nr}">
            </td>
      </c:forEach>
   </tr></table>
1  Ditt mobilnummer: <input type="text" name="kundemobil"><br>
   <input type="submit" value="Bestill">
</form>
</body>
</html>
```

Sensurnotater:

...

c) (5%) Skriv **kvittering.jsp**. *Tips1: Siden Bestilling er en nøstet struktur (inneholder et Map med antall av hver pizza) kan det kanskje være lurt/nødvendig å lagre detaljer underveis i page-attributter for ikke å få alt for komplekse EL-uttrykk. Tips2: Å hente en verdi fra et map m kan gjøres med m[key].*

Løsningsforslag:

Det vi prøver å generere er ca. noe slikt (dynamiske data merket med **fet blå**):

```
<html>
<body>
<h3>L-Pizza - Kvittering</h3>
      <b>Bestilling 17 des. 2021 kl. 18:28</b> <br>
      Kundemobil: 123 45 678 <br>
      - 3 stk. #3 Exotic á kr 179 = kr 537<br>
      - 1 stk. #1 Grandis á kr 149 = kr 149<br>
      <b>Totalt å betale = kr 686</b><br><br>
      Vi takker for din bestilling!
</body>
</html>
```

--- Løsning. Bestillingen kan nås via attributten "bestilling" fra 3a):

```
<html>
<body>
<h3>L-Pizza - Kvittering</h3>
1     <b>Bestilling ${bestilling.tidspunkt}</b> <br>
2     Kundemobil: <c:out value="${bestilling.kundemobil}"/> <br>
2     <c:forEach var="pizza" items="${bestilling.pizzaer}">
2           - ${bestilling.antallAv[pizza]} stk. #${pizza.nr} ${pizza.navn}
1           á kr ${pizza.pris} = kr ${bestilling.delsum[pizza]}<br>
      </c:forEach>
2     <b>Totalt å betale = kr ${bestilling.totalpris}</b><br><br>
      Vi takker for din bestilling!
</body>
</html>
```

Sensurnotater:

Se neste side ...

I tillegg til rene data fra bestillingsobjektet er behov for et par beregnete verdier, nemlig sum for hver rad/pizzatype og totalsum. Hvis vi tenker at disse skal hentes via ${} må det være rene get-metoder uten parametre, f.eks.:
 --- Map<Pizza, Integer> **getDelsum**()
 --- int **getTotalpris**()

Et alternativ kan være å gjøre utregningene i JSP-en og evt. mellomlagre i page-attributter inni løkken, f.eks:

```
<c:set var="delsum" value="${bestilling.antallAv[pizza] * pizza.pris}" />
... = kr ${delsum}<br>
```

I løsningsforslaget over itereres det over et **keyset**, altså en liste av pizzaene i bestillingen. Da blir det:

```
<c:forEach var="pizza" items="${b.keys}">
    ${pizza} gir pizza-objektet for ordrelinjen
    ${b.map[pizza]} gjør et oppslag som gir antallet for ordrelinjen
```

Alt2: Ser også at noen prøver å hente ut verdier fra et map på annen måte (.key og .value) enn beskrevet i oppgave/løsningsforslag. Dette gir mening hvis vi itererer over **entriene** i mappet uten oppslag:

```
<c:forEach var="pizzaAntall" items="${b.map}">
    ${pizzaAntall.key} gir pizza-objektet for ordrelinjen
    ${pizzaAntall.value} gir antallet for ordrelinjen
```

Alt3: Noen har også iterert over den fulle listen av pizzaer fra menyen. Da blir det noe slikt:

```
<c:forEach var="pizza" items="${meny}">
    <c:if test="${b.map[pizza] > 0}">
        ${pizza} gir pizza-objektet for ordrelinjen
        ${b.map[pizza]} gjør et oppslag som gir antallet for ordrelinjen
```

Dette delspørsmålet var ikke så bra besvart, selv på de beste besvarelsene. Ved poenggiving vil 3c) derfor gis vekt 2% i stedet for de 5% som står i oppgaveteksten (1p->a, 2p->b). Hvis enkelte studenter kommer dårligere ut etter omregning vil de kompenseres tilsvarende.

d) (5%) List opp alle kontruktører og metoder i klassen **Bestilling** som du har brukt i løsningen. Du trenger ikke å vise implementasjon, kun **returtype**, **metodenavn** og evt. **parametre m/typer**. Skriv gjerne en kort kommentar om det er uklart hva metoden gjør.

Løsningsforslag:

```
//Brukt i a), PizzamenyServlet.doPost():

      // Konstruktør. Både tidspunkt og tomt map for antall settes her.
      public Bestilling(String kundemobil)

      // Legger til et antall av en gitt pizza.
      public void leggTil(Pizza pizza, int antall)

//Brukt i c), kvittering.jsp:

      // Gettere
      public String getTidspunkt()                EL: ${b.tidspunkt}
      public String getKundemobil()               EL: ${b.kundemobil}
      public Map<Pizza, Integer> getAntallAv()  EL: ${b.antallAv[p]} for pizza p

      // Henter et Set av pizzaer i bestillingen.
      public Set<Pizza> getPizzaer()              EL: ${b.pizzaer}

      //Beregner alle delsummene (ordrelinjene).
      public Map<Pizza, Integer> getDelsum()      EL: ${b.delsum[p]} for pizza p

      // Beregner totalpris for hele bestillingen.
      public int getTotalpris()                   EL: ${b.totalpris}
```

Sensurnotater:

Dette spørsmålet ble i hovedsak gitt for å sjekke om dere forstår sammenhengen mellom objekters egenskaper/get-metoder og hva dette blir i EL.

I tillegg var Bestilling-klassen helt åpen og opp til dere å definere API for, så det kunne kanskje være greit å se hvilke valg dere har gjort.

For å få god score på dette delspørsmålet må API-et til Bestilling samsvare med det dere har gjort i a) og c). Spørsmålet sier "... **som du har brukt i løsningen**". Metoder du ikke har brukt i løsningen er ikke relevante. Hvis du ikke har brukt et Bestilling-objekt i det hele tatt er det vanskelig å gi særlig score på denne.

Hvis veldig mye logikk er flyttet inn i metoder her må de forklares godt. F.eks. hvis man har en konstruktør Bestilling(... request), så bør det foklares hvordan data trekkes ut av request. F.eks. hvis man har en metode getKvitteringAsHtml() som produserer hele kvitteringen, så bør den forklares/begrunnes godt. Man skal ikke kunne "tolke seg bort" fra det det blir spurt om.

e) (5%) La oss nå tenke oss at vi ønsker å kunne tilby denne pizzabestillings-appen på flere språk, og vi ønsker å bruke request-headeren **`Accept-Language`** til å bestemme hvilket språk vi skal velge. La oss videre tenke oss at vi har en ferdig hjelpeklasse **`SpraakUtil`** med hjelpemetoden **`finnBesteSpraak`** vi kan bruke til dette:

```
public class SpraakUtil {
      public static String finnBesteSpraak(HttpServletRequest request) { ... }
}
```

F.eks. skal en request med headeren **Accept-Language: fr, en;q=0.8, de;q=0.7** returnere **"fr"** som som foretrukket språk.

Hvordan kan man utføre **enhetstesting** for å teste at **`SpraakUtil.finnBesteSpraak(..)`** med headeren i eksemplet over gir **"fr"** som svar. For å få full score må du skrive en fungerende test.

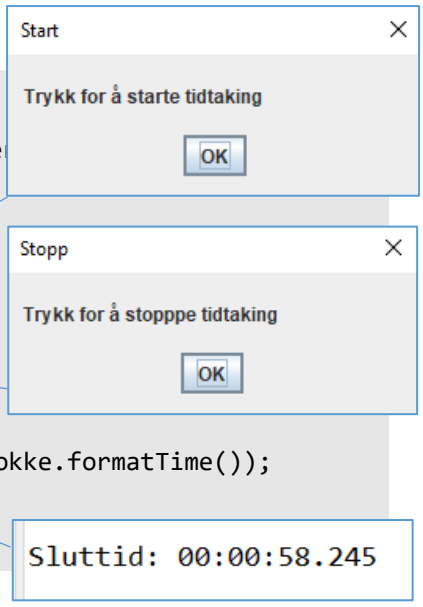
```
Løsningsforslag:

For å lage et request-objekt for anledningen kan vi f.eks. bruke
org.springframework.mock.web.MockHttpServletRequest.

class SpraakUtilTest {

    private MockHttpServletRequest request;

    @BeforeEach
    public void setup() {
        request = new MockHttpServletRequest();
    }

    @Test
    void foretrukketSpraakVelges() {
        request.addHeader("Accept-Language", "fr, en;q=0.8, de;q=0.7");
        assertEquals("fr", SpraakUtil.finnBesteSpraak(request));
    }
}
```

```
Sensurnotater:

Det siste delspørsmålet på denne oppgaven regner jeg også som litt vanskelig. Vi
har gått gjennom dette på forelesning, og det er gitt tildsvarende
eksamensspørsmål tidligere, så noen burde nok klare denne likevel (håper jeg).

Resultat: Noen få klarte denne og fikk full score. De fleste hadde ikke svart.
```

Oppgave 4 (10% ~ 24 minutter) – Tråder

Vi tar utgangspunkt i programmet under. Her bruker vi et stoppeklokke-objekt til å ta tiden, og vi bruker JOptionPane til å starte og stoppe stoppeklokken.

```
public class StoppeklokkeMain {

	public static void main(String[] args) throws Inte

		StopWatch stoppeklokke = new StopWatch();

		JOptionPane.showConfirmDialog(...);
		stoppeklokke.start();

		JOptionPane.showConfirmDialog(...);
		stoppeklokke.stop();

		System.out.println("\nSluttid: " + stoppeklokke.formatTime());
	}
}
```



Vi har lyst å fikse dette programmet slik at det i tillegg, mens stoppeklokken går, kontinuerlig (f.eks. hvert 10. millisekund) viser tiden så langt. (NB! Dette må skje samtidig som vi venter på at bruker skal stoppe tidtakingen.)

Ved å bruke **System.out.print("\r"** + det som skal skrives ut**)** i en løkke kan vi skrive på samme linje i konsollet om igjen og om igjen uten at vi hopper til neste linje. *(Antar at "\r" (=return) hopper til starten av linjen uten å hoppe ned slik "\n" (=newline) gjør.)*

```
Sensurnotater:

Det er flere mulige løsningsvarianter på denne oppgaven, f.eks. i hvilken grad
StopWatch integreres med utskriftstråd. Her kan man ha alt fra at de nesten kjører
uavhengig og kontrolleres separat i main, til en løsning der styringen av
StopWatch er helt innkapslet i tråden, evt. at tråden arver fra StopWatch.

Jeg ser at en del studenter "implementerer" sin egen variant av StopWatch.
Meningen var å endre/utvide på eksisterende løsning, ikke lage en alternativ
StopWatch-klasse. Dette burde kanskje kommet tydeligere frem i oppgaveteksten. Kan
nok ikke trekke så mye for det i utgangspunktet, kun hvis løsningen inneholder
feil. Noen antar også at stoppeklokken er org.apache.commons.lang3.time.StopWatch,
og bruker metoder fra denne i løsningen. Det kan man jo ikke anta!

Et poeng i oppgaven er å stoppe tråden på en kontrollert måte, dvs. ikke bruke
Thread.stop() som er deprecated. Man kan også argumentere for at det bør gjøres en
join() på tråden før sluttiden vises for å være sikker på at utskriftsløkken og
run() er avsluttet.

Nyttig merknad fra en student: Det er en bug at "\r" ikke fungerer i eclipse. Det
var lagt til en instilling for å ordne dette i eclipse 2020-03: Window ->
Preferences -> Run/Debug -> Console -> Interpret ASCII control characters
```

Oppgave: Gjør de nødvendige endringer i programmet + lag evt. nye interfaces og klasser du trenger for å få dette til. *Tips: Det er sikkert lurt å lage en trådklasse.*

```
Løsningsforslag, alternativ1 med StopWatch og tråd hver for seg.
Den delen av løsningen som er merket med grått er uforandret fra oppgaveteksten.

public class ViseTidTraad extends Thread {

	private boolean viseTid = true;
	private StopWatch stoppeklokke;

	public ViseTidTraad(StopWatch stoppeklokke) {
		this.stoppeklokke = stoppeklokke;
	}

	public void stopp() {
		viseTid = false;
	}

	@Override
	public void run() {
		while(viseTid) {
			System.out.print("\r\t" + stoppeklokke.formatTime());
			try {
				Thread.sleep(10);
			} catch (InterruptedException e) {
			}
		}
	}
}
public class StoppeklokkeMain {

	public static void main(String[] args) throws InterruptedException {

		StopWatch stoppeklokke = new StopWatch();

		ViseTidTraad viseTidTraad = new ViseTidTraad(stoppeklokke);
		viseTidTraad.start();

		JOptionPane.showConfirmDialog(...);
		stoppeklokke.start();

		JOptionPane.showConfirmDialog(...);
		stoppeklokke.stop();

		viseTidTraad.stopp();
		viseTidTraad.join();

		System.out.println("\nSluttid: " + stoppeklokke.formatTime());
	}
}
```

```
Løsningsforslag, alternativ2 med en "trådklasse" som extender StopWatch.
Den delen av løsningen som er merket med grått, dvs. i praksis hele main(), er
uforandret fra oppgaveteksten.

public class StopWatchSomViserTiden extends StopWatch implements Runnable {

	private boolean viseTid = true;
	Thread viseTidTraad = new Thread(this);

	@Override //StopWatch sin start()
	public void start() {
		viseTidTraad.start();	//Starte utskriftstråd
		super.start();		//Starte stoppeklokken
	}

	@Override //StopWatch sin stop()
	public void stop() {
		super.stop();		//Stoppe stoppeklokken
		viseTid = false;	//Stoppe utskriftstråd
	}

	@Override //Runnable sin run()
	public void run() {
		while(viseTid) {
			System.out.print("\r\t" + formatTime());
			try {
				Thread.sleep(10);
			} catch (InterruptedException e) {
			}
		}
	}
}
```

```
public class StoppeklokkeMain {

	public static void main(String[] args) throws InterruptedException {

		StopWatch stoppeklokke = new StopWatchSomViserTiden();

		JOptionPane.showConfirmDialog(...);
		stoppeklokke.start();

		JOptionPane.showConfirmDialog(...);
		stoppeklokke.stop();

		System.out.println("\nSluttid: " + stoppeklokke.formatTime());
	}
}
```

```
Andre alternative løsninger er også mulig.
```

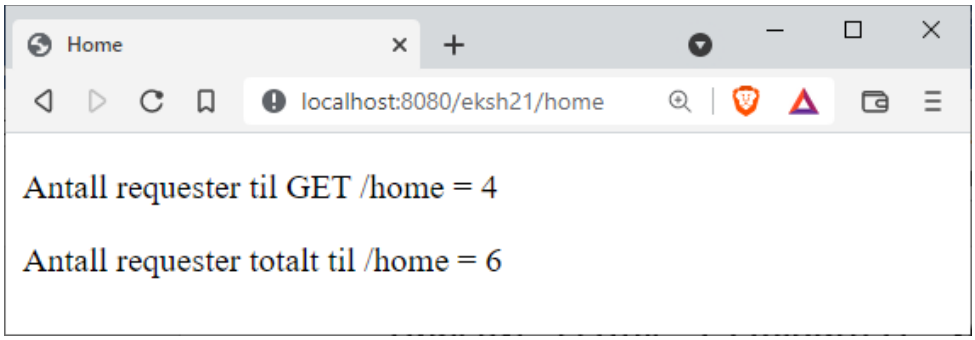
Oppgave 5 (10% ~ 24 minutter) – Web og tråder

Som dere vet vil en Java web-container (f.eks. Tomcat) opprette en ny tråd for hver request og kjøre doGet() eller doPost() i denne tråden. Tenk at vi har en applikasjon med én Servlet (HomeServlet) med doGet() og doPost().

Vi ønsker å holde orden på

1. Hvor mange requester det har vært til GET /home (doGet() i HomeServlet) siden oppstart
2. Hvor mange requester det har vært totalt (både GET og POST /home) siden oppstart

Begge tellerne skal kunne hentes frem i tilhørende JSP-side med expression language.



Oppgave: Skriv en løsning for hvordan disse to tellerne kan implementeres slik at de teller korrekt og på en trådsikker måte. Altså skriv det som er relevant i HomeServlet.

Løsningsforslag:

(Hadde vi hatt flere Servlets hadde nok det greieste være å lagre tellerne direkte i ServletContext-en, men ...)

Siden vi kun har én Servlet kan vi velge å ha tellerne som instans-/objekt-variabler i Servleten. I doGet() må vi da samtidig lagre verdiene i attributter i f.eks. request for tilgang fra jsp-en.

For å få det trådsikkert må oppdatering av teller(e) merkes som kritisk seksjon med synchronized.

```
@WebServlet("/home")
public class HomeServlet extends HttpServlet {

1       Integer antallGet = 0;
1       Integer antallPost = 0;

        @Override
        protected void doGet(...) {

2 1             synchronized(antallGet) {
                    antallGet++; }

2               request.setAttribute("antallGet", antallGet);
1               request.setAttribute("antallTotalt", antallGet + antallPost);

                //Alle de andre tingene som skal gjøres i doGet() ...

                request.getRequestDispatcher("WEB-INF/home.jsp").forward(...);
        }

        @Override
        protected void doPost(...) {

1 1             synchronized(antallPost) {
                    antallPost++; }

                //Alle de andre tingene som skal gjøres i doPost() ...

                response.sendRedirect("home");
                //NB! Merk at redirect fører til enda en GET-request.
                //    Hvis dette ikke er ønskelig kunne man telt på en
                //    annen måte enn over, nemlig antallGet--;
        }
}
```

Sensurnotater:

Se neste side ...

Sensurnotater:

Siden dette er siste oppgaven i settet regner jeg med at ikke alle har fått tid til å gjøre denne. Vi får se hva vi gjør med dette i sensuren, f.eks. om det viser seg at mange har hatt tidsnød.

En alternativ løsning kan være å bruke AtomicInteger.

```
@WebServlet("/home")
public class HomeServlet extends HttpServlet {

	AtomicInteger antallGet = new AtomicInteger();
	AtomicInteger antallPost = new AtomicInteger();

	@Override
	protected void doGet(...) {

		request.setAttribute("antallGet", antallGet.incrementAndGet());
		request.setAttribute(
			"antallTotalt", antallGet.get() + antallPost.get());

		//Alle de andre tingene som skal gjøres i doGet() ...

		request.getRequestDispatcher("WEB-INF/home.jsp").forward(...);
	}

	@Override
	protected void doPost() {

		antallPost.incrementAndGet();

		//Alle de andre tingene som skal gjøres i doPost() ...

		response.sendRedirect("home");
		//NB! Merk at redirect fører til enda en GET-request.
		//    Hvis dette ikke er ønskelig kunne man telt på en
		//    annen måte enn over, nemlig antallGet.decrementAndGet();
	}
}
```