

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgave i TDT4100 **Objektorientert programmering**

Faglig kontakt under eksamen: Hallvard Trætteberg

Tlf.: 918 97 263

Eksamensdag: Tirsdag 16. mai

Eksamenstid (fra-til): 9.00-13.00

Hjelpemiddelkode/Tillatte hjelpemidler: C

Kun "Big Java", av Cay S. Horstmann, er tillatt.

Annen informasjon:

Oppgaven er utarbeidet av faglærer Hallvard Trætteberg og kvalitetssikret av Ragnhild Kobro Runde (Ifi, UiO).

Målform/språk: Bokmål

Antall sider: 4

Antall sider vedlegg: 8 (5+1+2)

Kontrollert av:

Dato

Sign

Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger som du finner nødvendig. Hvis du i en del er bedt om å *implementere* klasser og metoder og du ikke klarer det (helt eller delvis), så kan du likevel *bruke* dem i senere deler.

En oversikt over klasser og metoder for alle oppgavene er gitt i vedlegg 1. Kommentarene inneholder krav til de ulike delene, som du må ta hensyn til når du løser oppgavene. I tillegg til metodene som er oppgitt, står du fritt til å definere ekstra metoder for å gjøre løsningen ryddigere. Nyttige standardklasser og -metoder finnes i vedlegg 3.

Temaet for oppgaven er et spisested (**Diner**) og problemstillingen er plassering (**Seating**) av grupper (**Group**) av gjester ved bordene (**Table**).

Del 1 – Group, Table- og Seating-klassene (15%)

Group-, **Table**- og **Seating**-klassene (vedlegg 1) er såkalte verdi-klasser, med data som skal oppgis ved opprettelsen av objektene og siden ikke skal kunne endres. **Group** skal inneholde data om antall gjester i gruppa, **Table** skal inneholde data om antall stoler (**capacity**) og **Seating** skal holde rede på bordet en gitt gruppe sitter på.

- a) Skriv ferdig **Group** og **Seating**, inkludert nødvendige innkapslingsmetoder.

```
public class Group {  
  
    private final int guestCount;  
  
    public Group(int guestCount) {  
        this.guestCount = guestCount;  
    }  
  
    public int getGuestCount() {  
        return guestCount;  
    }  
}  
  
public class Seating {  
  
    private final Table table;  
    private final Group group;  
  
    public Seating(Table table, Group group) {  
        this.table = table;  
        this.group = group;  
    }  
  
    public Table getTable() {  
        return table;  
    }  
  
    public Group getGroup() {  
        return group;  
    }  
}
```

- b) En skal ikke kunne ha **Seating**-objekter for bord som ikke har mange nok stoler til hele gruppa som er plassert der. Skriv koden som trengs for å sikre at denne regelen overholdes.

Følgende kode legges øverst i konstruktøren:

```
if (table.getCapacity() < group.getGuestCount()) {  
    throw new IllegalArgumentException("The table is too small for the group");  
}
```

- c) Anta at **Group** hadde en metode for å endre antall gjester. Forklar med tekst og/eller kode hvilke endringer du måtte gjort for å sikre at regelen i b) overholdes.

Group måtte hatt en referanse til **Seating**-objektet (eller **Table**-objektet) som ble satt av **Seating**, så den kunne kjøre tilsvarende sjekk som over, ved endring av størrelsen.

- d) I tillegg til antall stoler, skal et bord ha et bordnummer. Dette skal være et unikt løpenummer som ikke oppgis, men settes automatisk av kode i **Table**-klassen selv når **Table**-objekter opprettes. Det aller første bordet som lages skal få 1 som nummer, det andre skal få 2 osv. Implementer konstruktøren og annen nødvendig kode, inkludert **getNum**-metoden!

Her er poenget at en trenger en *global* teller, som en får til i Java ved bruk av **static**. Denne må brukes og økes i **Table** sin konstruktør.

```
private static int tableCounter = 1;  
  
private final int num;  
private final int capacity;  
  
public Table(int capacity) {  
    this.num = tableCounter++;  
    this.capacity = capacity;  
}
```

Del 2 – Diner-klassen (40%)

Diner-klassen (vedlegg 1) holder rede på bord (**tables**) og bordplasseringer (**seatings**), altså hvilke grupper som sitter ved hvilke bord.

- a) Skriv nødvendige felt-deklarasjoner og konstruktør(er), gitt at spisestedet har mer enn ett bord. Skriv også metodene for å legge til og fjerne bord.

Vi velger **Collection**, fordi vi ikke trenger indeksbasert tilgang til elementene. Vi ser ikke noe behov for en konstruktør, siden en like gjerne kan legge til bord ved kall til **addTable**. Det er greit nok å ha en konstruktør som tar inn et sett bord, f.eks. en varargs som i **Table... tables**, og da er det viktig å initialisere lokale variables slik at de ikke kan endres utenifra. Merk at **Arrays.asList(...)** lager en **List** som ikke kan endre størrelse, så den kan ikke bruke direkte som verdien til et lokal **tables**-felt. Det vil være mulig å bruke én **tableSeatings-Map/HashMap** i stedet for to lister, hvor **tableSeatings.keySet()** tilsvare **tables** og **tableSeatings.values()** tilsvare **seatings**.

```
private Collection<Table> tables = new ArrayList<>();  
private Collection<Seating> seatings = new ArrayList<>();  
  
public void addTable(Table table) {  
    tables.add(table);  
}
```

```

public void removeTable(Table table) {
    if (isOccupied(table)) {
        throw new IllegalArgumentException("Cannot remove an occupied table");
    }
    tables.remove(table);
}

```

b) Skriv metodene **isOccupied** og **getCapacity**.

```

public boolean isOccupied(Table table) {
    for (Seating seating : seatings) {
        if (seating.getTable() == table) {
            return true;
        }
    }
    return false;
    // return seatings.stream().anyMatch(s -> s.getTable() == table);
}

public int getCapacity(boolean includeOccupied) {
    int capacity = 0;
    for (Table table : tables) {
        if (includeOccupied || (! isOccupied(table))) {
            capacity += table.getCapacity();
        }
    }
    return capacity;
    // Stream<Table> stream = tables.stream();
    // if (! includeOccupied) {
    //     stream = stream.filter(t -> ! isOccupied(t));
    // }
    // return stream.mapToInt(Table::getCapacity).sum();
}

```

c) Bord kan settes samme, typisk for å få plass til store grupper med gjester. Tilsvarende kan bord deles opp, for å unngå at en liten gruppe tar opp et stort bord. Skriv metodene **mergeTables** og **splitTable**. I denne omgang skal det ikke registreres hvilke bord som faktisk settes sammen, de forsvinner bare, og må opprettes på nytt ved oppdeling.

isOccupied-sjekken kan virke unødvendig, siden den også gjøres i **removeTable**, men det er best å sjekke på forhånd og ikke underveis, så en ikke utfører endringene delvis (uten å reversere endringene).

```

public void mergeTables(Table table1, Table table2, int lostCapacity) {
    checkNotOccupied(table1);
    checkNotOccupied(table2);
    Table table = new Table(table1.getCapacity() + table2.getCapacity() - lostCapacity);
    removeTable(table1);
    removeTable(table2);
    addTable(table);
}

public void splitTable(Table table, int capacity1, int capacity2) {
    checkNotOccupied(table);
    removeTable(table);
    addTable(new Table(capacity1));
    addTable(new Table(capacity2));
}

```

```

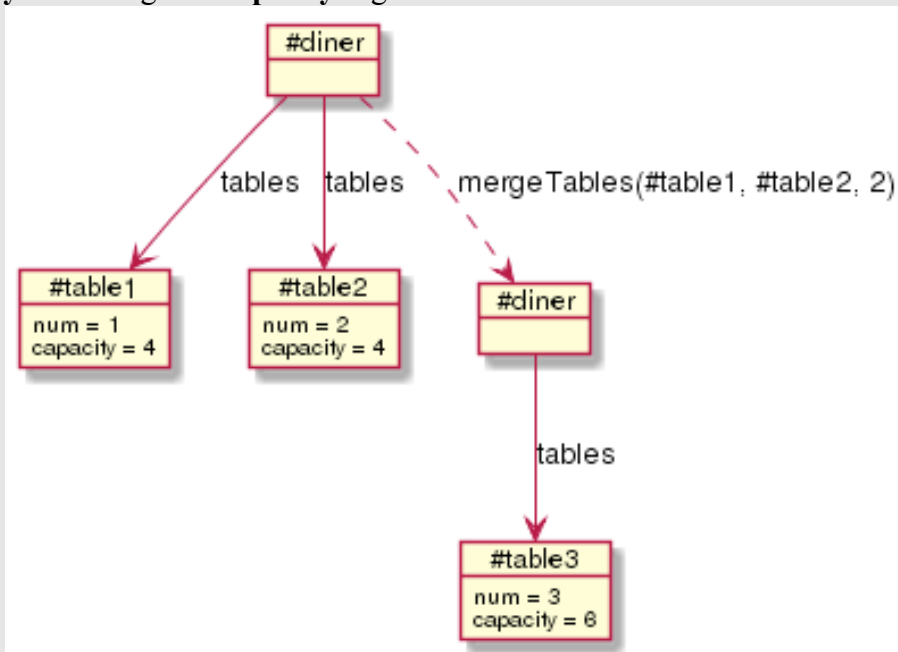
}

void checkNotOccupied(Table table) {
    if (isOccupied(table)) {
        throw new IllegalArgumentException("The table cannot be occupied");
    }
}
}

```

d) Tegn et objekttilstandsdiagram som illustrerer virkemåten til **mergeTables**.

Den første tilstanden kan inneholde **Diner**-objekt koblet til to **Table**-objekter. Transisjonen er et kall til **mergeTables**. Den andre tilstanden inneholder det samme **Diner**-objektet koblet til et nytt **Table**-objekt. **capacity**-feltene og **lostCapacity**-argumentet må stemme overens.



Objektene og **mergeTables**-kallet har konkrete verdier, og de to **Table**-objektene er koblet til et **Diner**-objekt før kallet, og et nytt **Table**-objekt er koblet til (det logisk sett samme) **Diner**-objektet etter kallet. **Capacity**-attributtene må være med, men **num**-attributtet er ikke så viktig, så lenge det går frem at **Table**-objektet tilsvarende **table3** er nyopprettet.

e) Når gjester skal plasseres må en finne det minste, ledige bordet med nok kapasitet. Skriv metodene **hasCapacity** og **findAvailableTables**. Skriv også annen nødvendig kode for å tilfredsstille kravet om sortering av returverdien til **findAvailableTables**.

For **hasCapacity** er det viktig å ha med en sjekk for om bordet er opptatt, ikke bare sjekke bordets totale kapasitet. For **findAvailableTables** så er litt av poenget å skjønne at selv om en **Collection** ikke alltid har en ordning og kan sorteres, så kan en lage og returnere en **List** som er ordnet og sortert.

```

public boolean hasCapacity(Table table, int capacity) {
    return (! isOccupied(table)) && table.getCapacity() >= capacity;
}

public Collection<Table> findAvailableTables(int capacity) {
    List<Table> result = new ArrayList<>();
    for (Table table : tables) {
        if (hasCapacity(table, capacity)) {
            result.add(table);
        }
    }
}

```

```

    }
    Collections.sort(result);
    return result;
}

```

Table må implementere **Comparable<Table>** for at **sort**-metoden skal kunne brukes og virke. Alternativt kan man lage en **Comparator<Table>**, f.eks. med **(t1,t2) -> t1.getCapacity() - t2.other.getCapacity()**.

```

@Override
public int compareTo(Table other) {
    return getCapacity() - other.getCapacity();
}

```

- f) En ny bordplassering registreres i et **Seating**-objekt. Skriv metodene **createSeating**, **addSeating** og **removeSeating**.

Her er litt av poenget hvordan en skal håndtere at resultatet fra **findAvailableTables** er en **Collection**, som ikke har en "opplagt" metode for å hente ut første element. Her brukes **iterator()** og ett kall til **next()**. Kunne brukt **iterator()** først og så **hasNext()** for å sjekke om det er elementer å hente, i stedet for **isEmpty** (se utkommentert kode). Det er ikke bra å caste til **List**, selv om en vet at det er en **List** i praksis, eller endre på retur-typen til **findAvailableTables**.

Det er ikke meningen å utlyse unntak i **createSeating**, hvis ingen bord er ledige, men hvis det gjøres så må **addSeating** kodes med **try/catch** og riktig retur-verdi.

```

public Seating createSeating(Group group) {
    Collection<Table> availableTables = findAvailableTables(group.getGuestCount());
    if (availableTables.isEmpty()) {
        return null;
    }
    return new Seating(availableTables.iterator().next(), group);
// Iterator<Table> tablesIt = availableTables.iterator();
// return (tablesIt.hasNext() ? new Seating(tablesIt.next(), group) : null;
}

public boolean addSeating(Group group) {
    Seating seating = createSeating(group);
    if (seating != null) {
        seatings.add(seating);
        return true;
    }
    return false;
}

public void removeSeating(int tableNum) {
    for (Seating seating : seatings) {
        if (seating.getTable().getNum() == tableNum) {
            seatings.remove(seating);
            return; // eller break
        }
    }
}
}

```

Del 3 – Table, SimpleTable og CompositeTable (15%)

Et problem ved sammensetting og oppdeling av bord er at bordnummeringen blir gal, når logisk sett samme bord opprettes på nytt og får nytt nummer. En måte å håndtere det på er å ha to bord-typer, enkeltbord (**SimpleTable**) og sammensatt bord (**CompositeTable**), hvor sistnevnte holder rede på hvilke bord som er satt samme. Den nye versjonen av **mergeTable**-metoden må altså opprette et **CompositeTable** som inneholder de to bordene som settes sammen, og **splitTable**-metoden må skrives om så det deler opp et **CompositeTable** i de *samme* to bordene. **splitTable**-metoden trenger da ikke lenger de to **capacity**-argumentene fordi de to bordene jo vet sin kapasitet.

- a) Forklar med tekst og kode hvordan du vil bruke arv og/eller grensesnitt, slik at **Table** fortsatt kan brukes som generell bord-type og **SimpleTable** og **CompositeTable** kan håndtere hver sine spesialtilfeller. Forklar kort virkemåten til **SimpleTable** og **CompositeTable**.

SimpleTable og **CompositeTable** blir subclasser av **Table**, som selv enten blir et grensesnitt eller en abstrakt klasse med i hvertfall **getCapacity**-metoden. **SimpleTable** kan ta over det meste av **Table**-koden. **CompositeTable** kapsler inn informasjonen fra **mergeTable** og bruker de to **Table**-objektene til å beregne kapasiteten. Alternativt kan **Table** beholde **capacity**-feltet og **getCapacity()**-metoden og de to andre initialiserer **capacity** med **super(...)**. Vi har her ikke lagt opp til at **CompositeTable**-objekter skal få et løpenummer, men det er greit å la den funksjonaliteten være en del av en abstract **Table**-(super)klasse.

```
public class CompositeTable implements Table {

    private Table table1, table2;
    private int lostCapacity;

    public CompositeTable(Table table1, Table table2, int lostCapacity) {
        // alternativ: super(table1.getCapacity() + table2.getCapacity() - lostCapacity);
        this.table1 = table1;
        this.table2 = table2;
        this.lostCapacity = lostCapacity;
    }

    public Table getTable1() {
        return table1;
    }

    public Table getTable2() {
        return table2;
    }

    // alternativ: kan utelates om super(...) brukes over
    @Override
    public int getCapacity() {
        return table1.getCapacity() + table2.getCapacity() - lostCapacity;
    }
}
```

- b) Skriv nye versjoner av **Diner** sine **mergeTable**- og **splitTable**-metoder. Merk at den nye **splitTable**-metoden kun skal ta inn en **CompositeTable**.

```
public void mergeTables(Table table1, Table table2, int lostCapacity) {
    CompositeTable table = new CompositeTable(table1, table2, lostCapacity);
    removeTable(table1);
    removeTable(table2);
    addTable(table);
}
```

```

}

public void splitTable(CompositeTable table) {
    removeTable(table);
    addTable(table.getTable1());
    addTable(table.getTable2());
}

```

Del 4 – GuestManager (20%)

Gjester på en **Diner** tas i mot av en tilhørende **GuestManager** (se vedlegg 1), som prøver å plassere dem. Dersom det ikke går, så må de vente på at et bord med nok kapasitet blir ledig. **GuestManager** vil altså ha behov for å følge med på hvordan kapasiteten til **Diner**-objektet endres. Dette gjøres ved å gjøre **Diner** sin kapasitet-egenskapen, som returneres av et kall til **getCapacity(false)**, *observerbar*.

- a) Hva innebærer observerbarhet? Forklar kort med tekst og/eller kode hvordan en gjør en (egenskap i en) klasse observerbar.

Klassen må administrere et sett med lyttere (objekter som implementerer et lyttergrensesnitt), vha. felt for Collection av lyttere og add/remove-metoder. Alle steder hvor tilstanden (til egenskapen) endres, må det skytes inn kode som sier fra til lytterne (kall på fire-metode, som går gjennom lytterne).

- b) Forklar med tekst og/eller kode hvordan du vil endre **Diner** slik at **GuestManager** kan lytte til endringer i **capacity**-egenskapen (ved å implementere **CapacityListener**-grensesnittet).

Capacity-egenskapen beregnes på bakgrunn av **tables**- og **seatings**-listene, og derfor må lytterne varsles hver gang disse endres (av **addTable**, **removeTable**, **addSeating** og **removeSeating**). Det skiller ikke mellom om kapasiteten øker eller minker, selv om det er økning som **GuestManager** er interessert i.

- c) Forklar med tekst og/eller kode hvordan du vil skrive **GuestManager**-klassen. Vi forventer ikke komplisert logikk for sammenslåing eller splitting av bord, men de som kom først skal fortrinnsvis få bord først.

GuestManager må ha en liste **Group**-objekter som fungerer som en kø. Ved ankomst med **groupArrived**, så prøves først å kalle **addSeating**. Hvis det ikke går, så legges **Group**-objektet i køen. Når gruppa drar, så brukes **removeSeating**. For å kunne tømme køen, må det lyttes på endringer i kapasiteten, altså må **GuestManager** implementere **CapacityListener** og legge seg til som lytter på **Diner**-objektet. I **capacityChanged**-metoden må en gå gjennom køen og igjen prøve **addSeating** og evt. fjerne **Group**-objektet fra køen, hvis det gikk greit.

Eksempelkode:

```

public class GuestManager implements CapacityListener {

    private final Diner diner;

    public GuestManager(Diner diner) {
        this.diner = diner;
        diner.addCapacityListener(this);
    }

    private Collection<Group> waitingGroups = new ArrayList<>();
}

```



```

public void groupArrived(Group group) {
    if (! diner.addSeating(group)) {
        waitingGroups.add(group);
    }
}

public void groupDeparted(int tableNum) {
    diner.removeSeating(tableNum);
}

@Override
public void capacityChanged(Diner diner) {
    Iterator<Group> it = waitingGroups.iterator();
    while (it.hasNext()) {
        Group waiting = it.next();
        if (diner.addSeating(waiting)) {
            it.remove();
        }
    }
}
}

```

Del 5 – Diverse (10%)

- a) Er **CapacityListener** et funksjonelt grensesnitt? Begrunn svaret!

CapacityListener-grensesnitt er (teknisk sett) funksjonelt, siden det bare har én (abstrakt) metode (og kan derfor implementeres med lambda-syntaksen). Dette kreves i et svar som får poeng i det hele tatt. I tillegg bør andre argumenter (for at grensesnittet ikke er funksjonelt) trekkes inn, f.eks. at metoden typisk ikke er implementasjonens primære funksjon og at en ikke tenker på den som en matematisk funksjon som kun er avhengig av argumentene.

- b) Skriv (om) en av **isOccupied** eller **getCapacity** i **Diner** slik at den bruker **Stream**-teknikken og Java 8 sin funksjonssyntaks (hvis du ikke har gjort det fra før, da!).

```

public boolean isOccupied(Table table) {
    return seatings.stream().anyMatch(s -> s.getTable() == table);
}

public int getCapacity(boolean includeOccupied) {
    Stream<Table> stream = tables.stream();
    if (! includeOccupied) {
        stream = stream.filter(t -> ! isOccupied(t));
    }
    return stream.mapToInt(Table::getCapacity).sum();
}

```

- c) Forklar med tekst og/eller kode hvordan du vil teste **isOccupied**-metoden til **Diner** i en separat **DinerTest**-klasse, spesielt hvilke metoder i **Diner** du vil bruke. Angi om **Diner** evt. må endres for å være mer testbar.

Man må rigge opp en **Diner** med minst to bord og én **Seating**, og så sjekke at **isOccupied** returnerer riktig verdi for hvert bord. Opprigging er enklere og testen mer spisset om en får mulighet til å legge inn **Seating**-objekter uten å måtte bruke **addSeating(Group)**, så derfor kan en legge til en pakkeprivat **addSeating(Seating)**-metode i **Diner**.

Appendix 1: Provided code (fragments)

```
// part 1

/**
 * A group (of people) dining together, and should be seated at the same table.
 * We currently only need to handle the size.
 */
public class Group {

    /**
     * Initializes this Group with the provided guest count
     */
    public Group(int guestCount) {
        ...
    }
}

/**
 * A table with a certain maximum capacity.
 */
public class Table {

    /**
     * Initializes this Table with the provided capacity.
     * The table is also assigned a unique number.
     * @param capacity
     */
    public Table(int capacity) {
        ...
    }

    /**
     * @return the table number
     */
    public int getNum() {
        ...
    }
}

/**
 * Represents the fact that a Group is seated at and occupies a Table
 */
public class Seating {

    /**
     * Initializes this Seating ...
     */
    public Seating(...) {
        ...
    }
}
```

```

// part 2

/**
 * A place where groups of guests can buy a meal
 */
public class Diner {

    /**
     * Tells whether a Table is occupied.
     * @param table the Table to check
     * @return true if anyone is sitting at the provided Table
     */
    public boolean isOccupied(Table table) {
        ...
    }

    /**
     * Computes the guest capacity,
     * either the remaining (includeOccupied == false) or total (includeOccupied == true).
     * @param includeOccupied controls whether to include tables that are occupied.
     * @return the guest capacity
     */
    public int getCapacity(boolean includeOccupied) {
        ...
    }

    /**
     * Adds a table to this Diner
     * @param table
     */
    public void addTable(Table table) {
        ...
    }

    /**
     * Removes a Table from this Diner.
     * If the table is occupied an IllegalArgumentException exception should be thrown.
     * @param table
     * @throws IllegalArgumentException
     */
    public void removeTable(Table table) {
        ...
    }

    /**
     * Merges two tables, i.e. replaces two tables with one table.
     * lostCapacity is the difference between the old capacity and the new.
     * This number is typically positive, since seats are lost when moving two tables
     * close to each other.
     * @param table1
     * @param table2
     * @param lostCapacity
     * @throws IllegalArgumentException if any of the tables are occupied
     */
    public void mergeTables(Table table1, Table table2, int lostCapacity) {
        ...
    }

    /**
     * Splits a table into two, i.e. replaces one tables with two tables.
     * The two capacities are the capacities of the two new tables.
     * @param table
     * @param capacity1
     * @param capacity2

```

```

    * @throws IllegalArgumentException if the table is occupied
    */
    public void splitTable(Table table, int capacity1, int capacity2) {
        ...
    }

    /**
     * Tells whether a table has the provided capacity,
     * i.e. if that number of new guests can be seated there.
     * Note that a table cannot be shared among different groups.
     * @param table
     * @param capacity
     * @return true if capacity number of guests can be seated here, false otherwise.
     */
    public boolean hasCapacity(Table table, int capacity) {
        ...
    }

    /**
     * Returns the tables that has the provided capacity.
     * The tables should be sorted with the one with the least capacity (but enough) first.
     * @param capacity
     * @return the tables that has the provided capacity
     */
    public Collection<Table> findAvailableTables(int capacity) {
        ...
    }

    /**
     * Finds a suitable, existing table for the provided group, and creates
     * (but doesn't add) a corresponding Seating.
     * The chosen table should be the one with the least capacity.
     * @param group the group to be seated
     * @return the newly created Seating
     */
    public Seating createSeating(Group group) {
        ...
    }

    /**
     * Creates and adds a Seating for the provided group, using the createSeating method.
     * @param group
     * @return true if a Seating was created and added, false otherwise.
     */
    public boolean addSeating(Group group) {
        ...
    }

    /**
     * Removes the seating for the provided table (number), if one exists
     * @param tableNum the number of the table to be removed
     */
    public void removeSeating(int tableNum) {
        ...
    }
}

```

```

// part 3

public class SimpleTable ... Table {

    public SimpleTable(int capacity) {
        ...
    }

    ...
}

/**
 * A table that consists of two other tables.
 */
public class CompositeTable ... Table {

    public CompositeTable(Table table1, Table table2, int lostCapacity) {
        ...
    }

    ...
}

// part 4

/**
 * Interface for listening to changes in Diner capacity
 */
public interface CapacityListener {
    /**
     * Called to inform that a Diner has changed capacity
     * @param diner
     */
    public void capacityChanged(Diner diner);
}

/**
 * Handles guests arriving at and departing from a Diner.
 */
public class GuestManager ... {

    public GuestManager(Diner diner) {
        ...
    }

    /**
     * Handles arriving groups, by either seating them immediately
     * (if possible) or putting them in queue. Those enqueued will
     * be seated when the Diner's (change in) capacity allows.
     * @param group
     */
    public void groupArrived(Group group) {
        ...
    }

    /**
     * Handles departing groups, by removing their seating.
     * @param tableNum the table where the group was seated
     */
    public void groupDeparted(int tableNum) {
        ...
    }

    ...
}

```