

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgave i TDT4100 Objektorientert programmering

Faglig kontakt under eksamen: Hallvard Trætteberg

Tlf.: 91897263

Eksamensdato: 22. 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:

Antall sider vedlegg:

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.

Del 1 – Typer og funksjonelle grensesnitt (15%)

a) Typen til Java-uttrykk er basert på typen til deluttrykkene. F.eks. vil uttrykket **1 + 2** ha typen **int**, fordi **int + int** gir en **int**. Bestem og forklar typen til følgende uttrykk:

- "Java" + "eksamen"
- "Java" + "eksamen" == "sant"
- 1 / 2
- "0" + "123".charAt(5)
- "0123".charAt(0) - '0'

String + String gir **String**

String + String == String gir **boolean**

int / int gir **int**

String + char gir **String** (selv om koden kræsjer)

char - char gir **int**

b) Anta at en har følgende variabel-deklarasjon og initialisering:

```
Collection<String> strings = new ArrayList<String>().
```

Hva er sammenhengen mellom typen på venstre- og høyresiden av tilordningstegnet (=)?

Hvordan påvirker **String**-spesialiseringen (altså det som står mellom <>) bruken av **strings**-variabelen?

Typen på høyresiden må være den samme eller en subklasse (inkl. implementasjonsklasse, som her) av typen på venstresiden. Spesialiseringen må være den samme. **String**-spesialiseringen påvirker parametertyper og returtyper for **Collection**- og **ArrayList**-metodene. F.eks. vil **get** returnere **String** og **add** og **set**-metodene vil ta en **String** som parameter.

c) Hva er et *funksjonelt* grensesnitt?

Et funksjonelt grensesnitt har bare én abstrakt metode, og resultatet av å utføre metoden skal alltid være det samme for samme argumenter. Dette gjør at man kan tenke på implementasjonen som en matematisk funksjon. Det er også et poeng (men underordnet) at grensesnittet er ment å være den *primære* funksjonen til klassen som implementerer den. Ellers gir det ikke så mye mening å bruke anonyme klasser/lambda-uttrykk til å implementere grensesnittet. Et eksempel på dette er **Comparator**, som kun implementeres for å sammenligne argumentene. **Comparable**-derimot, implementeres av dataklasser og er derfor en sekundær funksjon, som det ikke er noe poeng å implementere som primærfunksjon.

d) Det funksjonelle grensesnittet **Predicate<T>** er definert som følger:

```
public interface Predicate<T> {  
    /**  
     * Evaluates this predicate on the given argument.  
     * Returns true if the input argument matches the predicate,  
     * otherwise false  
     */  
}
```

```
boolean test(T t);  
}
```

Anta vi har en **Person**-klasse med metodene **getGender()** (returnerer tegnet 'F' dersom personen er en kvinne, 'M' om personen er en mann og '\0' om kjønnet er ukjent) og **getAge()** (returnerer alderen). Skriv en metode **getMatchingPersons** som tar inn en **Collection** av **Person**-objekter og et **Predicate** (også for personer) og returnerer en ny **Collection** med de personene som tilfredsstillers betingelsen angitt av **Predicate**-argumentet. Vis hvordan metoden kan kalles med et predikat som sier om personen skal kalles inn til sesjon, dvs. *er 18 år og mann*.

Metoden deklarerer som **Collection<Person> getMatchingPersons(Collection<Person> persons, Predicate<Person> test)**. Koden kan skrives på (minst) to måter, enten som en én-linjer med **Stream**-teknikken eller med en løkke som tester og legger til en resultat-liste. Kallet gjøres enklest med lambda-notasjonen: **getMatchingPersons(persons, p -> p.getGender() == 'M' && p.getAge() == 18)**

Del 2 – Segment-, Path- og Trip-klassene (55%)

I denne oppgaven skal du implementere klasser og metoder for å representere (buss)ruter/turer og kunne estimere gjenværende tid for resten av turen. Du kan tenke deg at dette brukes av koden i en info.tavle, som informerer passasjerene om antatt ankomsttid, og en tjeneste for å få SMS en viss tid for ankomst (for de som skal hente passasjerer på desinasjonen).

En *strekning* består av en sekvens av *segmenter* med hver sin angitte *lengde (s)* og (*reise)tid (t)*. Fra lengden og tiden kan en beregne (gjennomsnitt)sfarten $v = s/t$. Tiden som knyttes til et segment kan være faktisk registrert reisetid, eller antatt reisetid (basert på historiske data), avhengig av hvilke data en velger å legge inn. Et eksempel med tre segmenter er vist i Figur 1 (*s* er km, mens *t* er sekunder):

Segment 1	Segment 2	Segment 3
s = 20	s = 80	s = 10
t = 1200	t = 3600	t = 550

Figur 1

En antar at farten innen et segment er jevn, og passer på å dele opp en strekning i segmenter ut fra denne antakelsen. En strekning fra én by til en annen vil kanskje være delt i tre segmenter (se Figur 1) én for veien ut av den ene byen, en for delen mellom og en for veien inn til den andre byen, siden hver av disse segmentene har helt ulike kjøreforhold og dermed gjennomsnittsfart. Dersom en vei har varierende fartsgrense, så kan en dele opp i kortere segmenter. Hvis antakelsen om jevn fart innen et segment er riktig, så kan en estimere tid mellom steder underveis på en strekning.

Du skal først implementere klasser for strekning og segment, som vi velger å kalle **Path** og **Segment**.

- Skriv først **Segment**-klassen, med en konstruktør som tar inn lengde og tid for segmentet og nødvendige felt. Det skal ikke være mulig å endre egenskapene etter opprettelsen. Lag også én **get**-metode for hver av de tre egenskapene **distance** (lengde), **duration** (tid) og **speed** (fart).

```
public class Segment {  
  
    private final double duration, distance;
```

```

public Segment(double distance, double duration) {
    this.distance = distance;
    this.duration = duration;
}

public double getDuration() {
    return duration;
}

public double getDistance() {
    return distance;
}

public double getSpeed() {
    return distance / duration;
}
}

```

- b) Implementer **Path**-klassen, slik at den inneholder en sekvens av **Segment**-objekter. Det skal være mulig å initialisere **Path**-objekter med ingen, én eller flere segmenter. Skriv også metoder som gjør det mulig å legge til segmenter og gå gjennom dem basert på indeks. Velg selv fornuftige metode-navn og signaturer.

Her var poenget både å finne passende datastruktur og innkapslingsmetoder. En trenger bare én add-metode, selv om det er vist to har. For å gå gjennom segmentene trenger en både én metode for **count/size** (ellers vet en ikke når iterasjonen skal stoppe) og én for å hente ut et element.

Det var veldig mange som lurte på formuleringen ”gå gjennom dem basert på indeks”, antageligvis fordi det ikke var sagt at dette var *innkapsling for andre klasser*, ikke til bruk internt i klassen.

```

private final List<Segment> segments;

public Path(Segment... segments) {
    this.segments = new
ArrayList<Segment>(Arrays.asList(segments));
}

public int getSegmentCount() {
    return segments.size();
}

public Segment getSegment(int i) {
    return segments.get(i);
}

public void addSegment(Segment segment) {
    segments.add(segment);
}
}

```

```
public void addSegment(double distance, double duration) {
    addSegment(new Segment(distance, duration));
}
```

- c) Tegn objektdiagram for strukturen av **Path**- og **Segment**-objekter tilsvarende strekningen vist i Figur 1.

Objektdiagram, ikke objekttilstandsdiagram eller klassediagram!

- d) Lag følgende metoder (eksemplene er alle med utgangspunkt i Figur 1):

- **double getDuration()** – returnerer (reise)tiden til hele strekningen. For tilstanden tilsvarende Figur 1 vil det være **1200 + 3600 + 550**.
- **double getDistance(Segment fromSegment, Segment uptoSegment)** – returnerer lengden fra og med **fromSegment** til, men ikke med, **uptoSegment**. Hvis **fromSegment** er **null**, så beregnes lengden fra starten av strekningen. Hvis **uptoSegment** er **null**, så beregnes lengden av resten av strekningen. Utløs et passende unntak hvis en eller begge argumentene ikke er en del av denne strekningen. Her er noen eksempler på argumenter og forventet resultat:

fromSegment	uptoSegment	resultat
Segment 1	Segment 3	20 + 80
null	Segment 2	20
Segment 2	null	80 + 10
null	null	20 + 80 + 10

- **Segment getSegmentAt(double distance, boolean next)** – returnerer segmentet som en er i etter at lengden **distance** er tilbakelagt (fra starten av strekningen). Dersom **distance** tilsvarer enden på et segment, så bestemmer **next**-argumentet om det er dette segmentet som skal returneres (**next** er **false**) eller evt. segmentet etter (**next** er **true**). Her er noen eksempler på argumenter og forventet resultat:

distance (s)	next	resultat
s = 0.0	false	null
s = 0.0	true	Segment 1
s = s1	false	Segment 1
s = s1	true	Segment 2
s > 20 og s < 20 + 80		Segment 2
s > 20 + 80 og s < 20 + 80 + 10		Segment 3
= 20 + 80 + 10	false	Segment 3
= 20 + 80 + 10	true	null

```
public double getDuration() {
    double duration = 0.0;
    for (Segment segment : segments) {
        duration += segment.getDuration();
    }
    return duration;
}
```

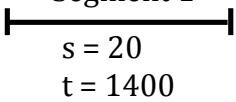
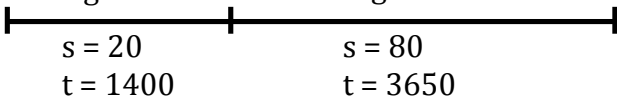
```

public double getDistance(Segment fromSegment, Segment uptoSegment) {
    if ((fromSegment == null || segments.contains(fromSegment)) &&
        (uptoSegment == null || segments.contains(uptoSegment))) {
        double distance = 0.0;
        for (Segment segment : segments) {
            if (segment == uptoSegment) {
                return distance;
            } else if (fromSegment == null || segment == fromSegment
|| distance > 0.0) {
                distance += segment.getDistance();
            }
        }
        return distance;
    } else {
        throw new IllegalArgumentException("Path doesn't contain both
segments.");
    }
}

public Segment getSegmentAt(double distance, boolean next) {
    for (Segment segment : segments) {
        if (distance == 0.0) {
            return (next ? segment : null);
        }
        distance -= segment.getDistance();
        if (distance < 0.0 || (distance == 0.0 && (! next))) {
            return segment;
        }
    }
    return null;
}

```

Klassen **Trip** representerer en faktisk (kjøre)tur og brukes for å oppdatere info.tavla på bussen med faktisk og estimert (reise)tid. **Trip**-klassen skal initialiseres med et **Path**-objekt som representerer *forventet* forløp, altså med segment-tider basert på historiske data. **Trip** skal også inneholde et **Path**-objekt med det *faktiske* forløpet, slik de registreres underveis på turen. Dette **Path**-objektet vil altså utvides med flere og flere **Segment**-objekter, etter hvert som segmentene passeres/tilbakelegges. Dette er illustrert i Figur 2:

<p>Første segment er registrert (med registerSegment-metoden, se under). Lengden er forventet lengde, men det har gått litt saktere.</p>	<p style="text-align: center;">Segment 1'</p>  <p style="text-align: center;">s = 20 t = 1400</p>
<p>Andre segment er registrert (med registerSegment). Her er både lengde og tid som forventet.</p>	<p style="text-align: center;">Segment 1' Segment 2'</p>  <p style="text-align: center;">s = 20 s = 80 t = 1400 t = 3650</p>

Tredje segment er registrert (med registerSegment). Her har det gått litt saktere enn forventet.	Segment 1'	Segment 2'	Segment 3'
	s = 20 t = 1400	s = 80 t = 3650	s = 10 t = 700

Figur 2

- e) Implementer nødvendige felt og konstruktør for **Trip**. Implementer også **registerSegment**:
- **void registerSegment(double distance, double duration)** – denne metoden kalles med en ny måling når et nytt segment er passert/tilbakelagt. Et nytt Segment-objekt skal legges inn med data tilsvarende målingen. Men merk at **distance**- og **duration**-argumentene er målt fra *starten* av hele turen, så de må regnes om før **Segment**-objektet opprettes. Hvis målt avstand ikke tilsvarer det en forventer ut fra *forventet* forløp eller det allerede er registrert nok segmenter, så skal det utløses et passende unntak.

```
private final Path path;

public Trip(Path path) {
    this.path = path;
}

private Path actualPath = new Path();

public void registerSegment(double distance, double duration) {
    if (actualPath.getSegmentCount() >= path.getSegmentCount()) {
        throw new IllegalArgumentException("Already registered
enough segments");
    }
    Segment pathSegment = path.getSegmentAt(distance, true);
    double expectedDistance = path.getDistance(null, pathSegment);
    if (expectedDistance != distance) {
        throw new IllegalArgumentException("Illegal distance,
should have been " + expectedDistance + ", but was " + distance);
    }
    double segmentDistance = expectedDistance -
path.getDistance(null, path.getSegmentAt(distance, false));
    double segmentDuration = duration - actualPath.getDuration();
    actualPath.addSegment(segmentDistance, segmentDuration);
}
}
```

- f) **Trip**-klassen skal kunne brukes til å estimere tiden som *gjenstår* fra og med et punkt underveis, basert på *forventet* forløp (som ble angitt ved opprettelsen av **Trip**-objektet). Merk at dette punktet kan godt være midt i et segment. Koden under er et forsøk på implementasjon. De to argumentene er tilbakelagt lengde og tid brukt så langt.

```
public double estimateTime(double distance, double duration) {
    double remainingTime = 0.0;
    for (Segment segment : this.getActualPath()) {
        distance -= segment.getDistance();
    }
}
```

```

    if (distance < 0) {
        // gjenværende tid for segmented en er kommet til
        remainingTime = distance / segment.getSpeed();
    } else if (remainingTime > 0.0) {
        // legg til gjenværende tid for etterfølgende segment
        remainingTime += segment.getDuration();
    }
}
return remainingTime;
}

```

Denne implementasjonen gjør et par antakelser om **Trip**- og **Path**-klassene, hvilke? Det er også plantet en liten logisk feil i koden, hvilken?

Koden bruker metoden **getActualPath()** fra **Trip**-klassen og den er ikke nevnt i oppgaven. Her står kallet på et sted hvor Java forventer en **Iterable** (eller en array). Typen må her være **Iterable<Segment>** sin løkkevariabelen er av typen **Segment**. Metode-navnet **getActualPath()** legger opp til at **Path**-klassen implementerer **Iterable<Segment>** og altså har en **iterator()**-metode som returnerer en **Iterator<Segment>**. Her er (til orientering) **Path**-koden:

```

class Path implements Iterable<Segment> {

    private final List<Segment> segments;

    @Override
    public Iterator<Segment> iterator() {
        return segments.iterator();
    }
}

```

Feilen som er plantet er at fortegnet på **remainingTime**-variabelen (i den første **if**-grenen) blir feil, når (vi vet at) **distance** er negativ. En må altså snu fortegnet med en minus i uttrykket på høyresida i tilordningen.

Det er også en annen feil, nemlig navngiving av metoden **getActualPath**. Algoritmen er basert på at en går gjennom forventet path, dvs. **path**-feltet i **Trip**, ikke **actualPath**. Derfor burde metoden het **getPath** evt. **getExpectedPath**. Dette er opplagt forvirrende og noe vi tar hensyn til ved sensur.

Poenget over med at **Path** må implementere **Iterable** er greit, men den logiske feilen ble vanskelig å finne pga. navnefeilen.

På toppen av dette er det feil i test-logikken inni, så her er det mange riktige svar! For ordens skyld: Her er den helt riktige implementasjonen, også uten den ene feilen som var plantet:

```

public double estimateTime(double distance, double duration) {
    double remainingTime = 0.0;
    for (Segment segment : this.getExpectedPath()) {
        distance -= segment.getDistance();
        // hvis vi har begynt å akkumulere
        if (remainingTime > 0.0) {
            // akkumuler
            remainingTime += segment.getDuration();
        }
    }
}

```



```

    } // hvis turen har kommet til dette segmentet
    else if (distance < 0) {
        // begynn å akkumulere
        remainingTime = segment.getSpeed() * (- distance);
    }
}
return remainingTime;
}

```

Del 3 – Arv, grensesnitt og delegering (15%)

- a) Metoden for å estimere tiden som er brukt over, tar ikke i betraktning tiden en har brukt så langt. Hvis trafikken i et segment går vesentlig saktere enn forventet, f.eks. pga. veiarbeid, så vil estimatet bli for lavt. Forklar med tekst og kode hvordan en kan bruke *arv* for å lage flere varianter av **Trip**-klassen, med ulike metoder for estimering.

Vi lager en abstrakt klasse basert på **Trip** og gjør **estimateTime** abstract:

```

public abstract class AbstractTrip {

    protected AbstractTrip(Path path) {
        this.path = path;
    }

    public abstract double estimateTime(double distance, double duration);
}

```

Så lar vi **Trip** arve fra denne og implementere **estimateTime** som over.

```

public class Trip extends AbstractTrip {

    protected Trip(Path path) {
        super(path);
    }

    @Override
    public double estimateTime(double distance, double duration) {
        ...
    }
}

```

Andre varianter vil gjøre det samme, men implementere **estimateTime** med annen logikk. Det er strengt tatt ikke nødvendig å ha en abstrakt klasse, en kan alternativt bare arve fra **Trip**.

- b) Gitt følgende deklarasjon av et grensesnitt for estimering av gjenværende tid:

```

public interface TimeEstimator {
    double estimateTime(Path path, Path actualPath,
        double distance, double duration);
}

```

```
}
```

Metoden **estimateTime** tar inn to **Path**-objekter, det første representerer *forventet* forløp, det andre faktisk forløp, slik det er registrert så langt. De to siste argumentene er total lengde kjørt og tid brukt.

Forklar med tekst og kode hvordan du kan lage en (eller flere) implementasjon(er) av dette grensesnittet og bruke *delegeringsteknikken* for å implementere **Trip**-klassen sin **estimateTime**-metode. Vurder hvor *fleksibel* denne teknikken er sammenlignet med bruken av arv i a).

```
public class DefaultTimeEstimator implements TimeEstimator {  
  
    @Override  
    public double estimateTime(Path path, Path actualPath, double  
distance, double duration) {  
        ...  
    }  
}  
  
public class Trip {  
  
    ...  
  
    private TimeEstimator timeEstimator = new DefaultTimeEstimator();  
  
    public void setTimeEstimator(TimeEstimator timeEstimator) {  
        this.timeEstimator = timeEstimator;  
    }  
  
    public double estimateTime(double distance, double duration) {  
        return timeEstimator.estimateTime(path, actualPath, distance,  
duration);  
    }  
}
```

Delegeringsteknikken er mer fleksibel enn arv, fordi en når som helst kan bytte ut **TimeEstimator**-objektet og dermed også endre oppførselen til **estimateTime**-metoden. Arv-mekanismen frigjøres dessuten til andre mer ”verdige” formål.

Del 4 – ArrivalNotifier-klassen (15%)

Anta at du skal lage en **ArrivalNotifier**-klasse som har ansvar for å sende SMS en viss tid før ankomst til destinasjonen. Dette kan være nyttig for de som skal plukke opp passasjerer når de kommer frem.

- a) **ArrivalNotifier**-trenger å lagre mobilnumre og hvor mange minutter tid i forkant før ankomst hvert enkelt nummer skal motta SMS. Du kan anta at mobilnumre kan lagres som **String**. Forklar med tekst og/eller kode hvordan du vil gjøre det.

En kan lage en egen klasse som kombinerer mobilnummer og antall minutter i forkant de skal få varsel, eller bruke en **Map<String, Integer>**.

- b) Etterhvert som tiden går og turen forløper så må SMS-ene sendes ut. Det er to ting som må trigge utsendelse av SMS-er, at tiden går og at **Trip**-objektet endres og dermed potensielt tidsestimatet. Hvilken generell teknikk kan brukes til for å informere **ArrivalNotifier**-objektet om disse endringen? Forklar med tekst og/eller kode hvordan du vil anvende denne teknikken her. Gjør de antagelser du finner nødvendig om hjelpeklasser knyttet til tid.

Den generelle teknikken er *observatør-observert*, som både brukes for å si fra om at tiden går og om at **Trip** er endret. Tiden håndteres ved å anta at det finnes en **Clock**-klasse, som kan si fra hvert minutt til et sett med lyttere. En kan f.eks. ha grensesnittet **ClockListener** med **minuttPassed()**-metoden og metodene **addClockListener** og **removeClockListener**. **Trip** må gjøres *observerbar*: 1) en må ha lese-metoder for alle relevante data og 2) en må kunne registrere lyttere (som implementerer et lyttergrensesnitt f.eks. kalt **TripChangedListener**) som får beskjed når objektet endres (f.eks. vha. en metode kalt **tripChanged**). 1) Det som endres her er sekvensen av registrerte segmenter, så det er naturlig å innføre metoder for å lese disse, f.eks. metoder tilsvarende de **Path** har. 2) En må ha en **Collection** av **TripChangeListener** og **add-** og **removeTripChangeListener**-metoder. **tripChanged**-metoden må kalles fra **registerSegment**-metoden, etter at et nytt segment er lagt til.