NTNU
Institutt for datateknikk
og informasjonsvitenskap

Eksamensoppgave i

TDT4100 – Objektorientert programmering

Lørdag 19. mai 2011, kl. 09:00 - 13:00

Oppgaven er utarbeidet av faglærer Hallvard Trætteberg og kvalitetssikrer Trond Aalberg.
Kontaktperson under eksamen Hallvard Trætteberg (mobil 918 97263)

Språkform: Bokmål
Tillatte hjelpemidler: C
Kun Java Pocket Guide, utgitt av O'Reilly forlag, er tillatt.

Sensurfrist: Mandag 11. juni.

Les oppgaveteksten nøye. Finn ut hva det spørres etter i hver oppgave.

Dersom du mener at opplysninger mangler i en oppgaveformulering, gjør kort rede for de antagelser og forutsetninger som du finner nødvendig.

Del 1 – Innkapsling (20%)

Gitt følgende klasse, som representerer et *tidspunkt* på dagen:

```
public class DayTime {

	public final int hours, minutes;

	public DayTime(int hours, int minutes) {
		this.hours = hours;
		this.minutes = minutes;
	}

	public String toString() {
		return hours + ":" + minutes;
	}
}
```

a) Hva betyr **final**-nøkkelordet slik det er brukt her? På hva slags måte(r) ivaretar denne klassen formålet med innkapsling og på hva slags måte(r) ikke?

b) Beskriv den generelle teknikken og navnekonvensjonen(e) for å representere og kapsle inn en enkel verdi, f.eks. tall eller objektreferanse, som skal kunne endres etter at objektet er opprettet.

c) I hva slags metode(r) brukes *unntak* ifm. innkapsling, og hvordan? Vis gjerne med et eksempel!

d) Mange klasser inneholder et List<X>-felt og definerer addX, removeX og andre metoder som tilsvarer og bruker metodene i List-grensesnittet. En alternativ løsning kunne vært å arve fra en List<X>-implementasjon, f.eks. ArrayList<X>. Hvorfor brukes aldri denne løsningen i praksis?

Del 2 – Klasser (50%)

I denne oppgaven skal du implementere klasser og metoder for å håndtere en dagsplan med tidsrom, f.eks. avtaler. Klassen oppgitt i oppgave 1 a) kan brukes som en del av implementasjonen din. Husk at alle metoder kan brukes i implementasjonen av andre.

a) Du skal implementere en klasse **TimeSlot** for å representere et *tidsrom* innenfor en dag, f.eks. en avtale. Et **TimeSlot**-objekt har en beskrivelse (tekst), et start- og sluttidspunkt (time og minutt) og en varighet (minutter). Ingen av disse dataene skal kunne endres etter at **TimeSlot**-objektet er opprettet. Du velger selv hvordan disse dataene skal representeres og hvilke hjelpemetoder som evt. trengs. Merk at varighet kan beregnes fra starttidspunkt og sluttidspunkt, evt. at sluttidspunkt kan beregnes fra starttidspunkt og varighet.

Implementer følgende konstruktører og metoder:
- **TimeSlot(String description, int hours, int minutes, int duration)**: initialiserer med oppgitt beskrivelse og starttidspunkt (timer og minutter) og varighet (minutter). F.eks. vil **new TimeSlot(”TDT4100-forelesning”, 10, 15, 105)** representere en TDT4100-forelesning i tidsrommet 10:15-12:00.

- **String toString()**: returnerer en **String** på formen **beskrivelse@start-slutt** hvor **start** og **slutt** er på formen **tt:mm** (altså to siffer pr. tall). Dersom **toString()**-metoden kalles på **TimeSlot**-objektet fra forrige punkt skal det gi **"TDT4100-forelesning@10:15-12:00"**.
- **DayTime getStartTime()**: returnerer starttidspunktet som et **DayTime**-objekt (se oppgave 1).
- **DayTime getEndTime()**: returnerer sluttidspunktet som et **DayTime**-objekt (se oppgave 1).
- **int getDuration()**: returnerer varighet i minutter

b) Implementer følgende metoder:
- **boolean contains(int hours, int minutes)**: returnerer om dette **TimeSlot**-objektet inneholder tidspunktet angitt med **hours** og **minutes**. Merk at sluttidspunktet regnes ikke som å være inneholdt i tidsrommet. Dette betyr at **new TimeSlot("…", 8, 0, 30).contains(8, 0)** skal gi **true**, mens **new TimeSlot("…", 8, 0, 30).contains(8, 30)** skal gi **false**.
- **boolean overlaps(TimeSlot timeSlot)**: returnerer om dette **TimeSlot**-objektet overlapper med det angitte **TimeSlot**-objektet, dvs. om det finnes et tidspunkt som begge inneholder.

c) **TimeSlot**-klassen skal støtte sortering. **TimeSlot**-objektet med tidligst starttidspunkt sorteres først, og dersom starttidspunktene er like, så skal det med tidligst sluttidspunkt sorteres først. Forklar og implementer nødvendig kode.

d) Du skal implementere en klasse **DayPlan**, for å holde oversikt over alle avtalene (altså **TimeSlot**-objekter) for en dag, bl.a. gi muligheten til å legge til og fjerne **TimeSlot**-objekter. Velg selv hvilke felt og evt. hjelpemetoder som trengs.

Implementer følgende metoder:
- **void addTimeSlot(TimeSlot timeSlot)**: legger det angitte tidsrommet til denne dagsplanen
- **void removeTimeSlot(TimeSlot timeSlot)**: fjerner det angitte tidsrommet fra denne dagsplanen
- **TimeSlot getTimeSlotAt(int hours, int minutes)**: returnerer det *tidligste* tidsrommet som inneholder tidspunktet angitt med **hours** og **minutes**, ellers null.

d) Implementer følgende to metoder for tidsplanlegging:
- **boolean containsOverlapping()**: returnerer om det finnes overlappende tidsrom i denne dagsplanen.
- **Collection<TimeSlot> getFreeTime()**: returnerer en samling **TimeSlot**-objekter som representerer *fritiden* en har i løpet av en dag, dvs. tidsrommene som denne dagsplanen *ikke* dekker.

For begge disse metodene kan det være lurt å definere hjelpemetoder for å gjøre løsningen ryddigere.

Del 3 – Arv og delegering (20%)

a) Du skal implementere støtte for TDT4100-forelesninger, som et spesielt tidsrom som alltid er fra **10:15** til **12:00**. Vis hvordan en klasse **TDT4100Lecture** kan implementere dette vha. arv fra **TimeSlot**.

b) Du skal implementere støtte for en dagsplan som alltid inneholder et **TDT4100Lecture**-objekt, dvs. en TDT4100-forelesning fra 10:15-12:00. Det skal ikke være mulig å fjerne TDT4100-forelesningen eller legge inn andre tidsrom som overlapper med den. Følgende kode illustrerer hvordan det skal virke:

```
DayPlan tuesday = new TDT4100DayPlan();
System.out.println(tuesday.getTimeSlotAt(10, 30));
// prints "TDT4100 lecture@10:15-12:00"
```

```
tuesday.addTimeSlot(new TimeSlot("Coffee break", 11, 30, 60));
// throws appropriate exception, since it overlaps with the TDT4100 lecture
```

Vis hvordan dette kan implementeres i en **TDT4100DayPlan**-klasse vha. arv fra **DayPlan**.

c) Det er ofte nyttig å kunne la én dagsplan bygge på eller inkludere en eller flere andre, f.eks. la dagsplanen for en bestemt tirsdag inkludere tirsdagsplanen som gjelder for hele semesteret (hvor bl.a. TDT4100-forelesningen ligger). Følgende kode illustrerer hvordan det skal virke:

```
DayPlan repeatingTuesday = new TDT4100DayPlan();
DelegatingDayPlan tuesday = new DelegatingDayPlan(repeatingTuesday);
System.out.println(tuesday.getTimeSlotAt(10, 30));
// prints "TDT4100 lecture@10:15-12:00" since tuesday logically includes the
TDT4100Lecture in repeatingTuesday

tuesday.addTimeSlot(new TimeSlot("Coffee break", 11, 30, 60));
System.out.println(tuesday.containsOverlapping());
// prints "true" since timeSlot in tuesday overlaps with TDT4100Lecture in
repeatingTuesday
```

Skisser med tekst og kode hvordan *delegeringsteknikken* og arv fra **DayPlan** kan brukes for å implementere denne oppførselen, inkludert hvordan du evt. vil modifisere **DayPlan** for å gjøre løsningen ryddigere.

Del 4 – Input/output (IO) (10%)

a) Hva er den grunnleggende forskjellen på input/output-klassene **InputStream**/**OutputStream** og deres subklasser ift. **Reader**/**Writer** og deres subklasser?

b) Ifm. input/output brukes en egen type unntak, hvilken? På hva slags måte påvirker det kode som driver med input/output?

c) Hvorfor må vi lukke input- og output-"strømmer" med **close()**-metoden når vi er ferdige med dem? Hvordan sikrer man at det skjer også i tilfelle unntak?

NTNU
Institutt for datateknikk
og informasjonsvitenskap

Exam for

TDT4100 – Object-oriented programming

Saturday 19. May 2012, 09:00 - 13:00

The exam is made by responsible teacher Hallvard Trætteberg and quality assurer Trond Aalberg.
Contact person during the exam is Hallvard Trætteberg (mobile 918 97263)

Language: English
Supporting material: C
Only Java Pocket Guide, published by O'Reilly, is allowed.

Deadline for results: Monday 11. June.

Read the text carefully. Make sure you understand what you are supposed to do.

If information is missing you must clarify what assumptions you find necessary.

Note the percentages for each part, so you use your time wisely.

Part 1 – Encapsulation (20%)

The following class represents a time within a day:

```
public class DayTime {

	public final int hours, minutes;

	public DayTime(int hours, int minutes) {
		this.hours = hours;
		this.minutes = minutes;
	}

	public String toString() {
		return hours + ":" + minutes;
	}
}
```

a) What does the **final** keyword mean, as used here? In what way(s) do(es) this class properly encapsulate the data and in what ways does it not?

b) Describe the general technique and naming conventions used for representing and encapsulating simple values, such as numbers and object references, that can be modified after the object has been created.

c) In what kind of method(s) are *exceptions* used for supporting encapsulation, and how? Please illustrate with an example!

d) Many classes include a List<X> field and defines addX, removeX and other methods that correspond to and use methods in the List interface. An alternative solution would be to inherit from a List<X>, like ArrayList<X>. Why is this solution never used in practice?

Part 2 – Classes (50%)

In this part you must implement classes and methods for managing a day plan containing time intervals , e.g. appointments. The class provided in part 1 a) can be used as part of your implementation. Remember that all methods can be used in the implementation of others.

a) You must implement a **TimeSlot** class for representing a time interval within a day, e.g. an appointment. A **TimeSlot** objekt has a description (tekst), start and end times (hours and minutes) and a duration (minutes). It should not be possible to modify any of these after the **TimeSlot** object has been created. You must decide how to represent the data and what utility methods you need. Note that the duration can be computed from the start and end times, or the end time can be computed from the start time and duration.

Implement the following constructors and methods:
- **TimeSlot(String description, int hours, int minutes, int duration)**: initializes with the provided description, start time (hours and minutes) and duration (minutes). E.g. **new TimeSlot(”TDT4100-forelesning”, 10, 15, 105)** will represent a TDT4100 lecture in the time interval 10:15-12:00.

- **String toString()**: returns a **String** of the form **description@start-end** where **start** and **end** are of the form **hh:mm** (i.e. two digits pr. number). If the **toString()** method is invoked/called on the **TimeSlot** object from the previous bullet**"TDT4100-forelesning@10:15-12:00"** should be returned.
- **DayTime getStartTime()**: returns the start time as a **DayTime** object (see part 1).
- **DayTime getEndTime()**: returns the end time as a **DayTime**-object (see part 1).
- **int getDuration()**: returns the duration in minutes

b) Implement the following methods:
- **boolean contains(int hours, int minutes)**: returns if this **TimeSlot** object includes the time provided by **hours** and **minutes**. Note that the end time is not considered to be included in the time interval. This means that **new TimeSlot("…", 8, 0, 30).contains(8, 0)** must evaluate to **true**, while **new TimeSlot("…", 8, 0, 30).contains(8, 30)** must evaluate to **false**.
- **boolean overlaps(TimeSlot timeSlot)**: returns if this **TimeSlot** object overlaps the provided **TimeSlot** object, i.e. if there exists a time that is included by both.

c) The **TimeSlot** class must support sorting. The **TimeSlot** object with the earliest start time must be appear first, and if the start times are the same, the one with the earliest end time must appear first. Explain and implement the necessary code.

d) You must implement a **DayPlan** class, for managing alle the appointments (i.e. **TimeSlot** objects) for a day, including adding and removing **TimeSlot** objects. You must decide which fields and utility methods that are needed.

Implement the following methods:
- **void addTimeSlot(TimeSlot timeSlot)**: adds the provided time interval to this day plan
- **void removeTimeSlot(TimeSlot timeSlot)**: removes the provided time interval from this day plan
- **TimeSlot getTimeSlotAt(int hours, int minutes)**: returns the *earliest* time interval that includes the time given by **hours** and **minutes**, or null otherwise.

d) Implements the following two methods for time planning:
- **boolean containsOverlapping()**: returns if there exists overlapping time intervals in this day plan.
- **Collection<TimeSlot> getFreeTime()**: returners a collection of **TimeSlot** objects that represents the *unscheduled time* of a day, i.e. the time intervals that this day plan does *not* include.
For both these method you may find it useful to define utility methods to make the solution more tidy.

Part 3 – Inheritance and delegation (20%)

a) You must implement support for TDT4100 lectures, as a special time interval always from **10:15** to **12:00**. Show how a **TDT4100Lecture** class can be implemented by inheriting from **TimeSlot**.

b) You must implement support for a day plan that always contains a **TDT4100Lecture** object, i.e. a TDT4100 lecture from 10:15-12:00. It should not be possible to remove this TDT4100 lecture or add other time intervals that overlap with it. The following code illustrates the desired behavior:

```
DayPlan tuesday = new TDT4100DayPlan();
System.out.println(tuesday.getTimeSlotAt(10, 30));
// prints "TDT4100 lecture@10:15-12:00"
tuesday.addTimeSlot(new TimeSlot("Coffee break", 11, 30, 60));
// throws appropriate exception, since it overlaps with the TDT4100 lecture
```

Show how a **TDT4100DayPlan** class can be implemented by inheriting from **DayPlan**.

c) It is often useful to let one day plan extend or include one or more others, e.g. let the day plan for a specific Tuesday include the Tuesday plan for the semester schedule (where you'll find the TDT4100 lecture). The following code illustrates the desired behavior:

```
DayPlan repeatingTuesday = new TDT4100DayPlan();
DelegatingDayPlan tuesday = new DelegatingDayPlan(repeatingTuesday);
System.out.println(tuesday.getTimeSlotAt(10, 30));
// prints "TDT4100 lecture@10:15-12:00" since tuesday logically includes the
TDT4100Lecture in repeatingTuesday

tuesday.addTimeSlot(new TimeSlot("Coffee break", 11, 30, 60));
System.out.println(tuesday.containsOverlapping());
// prints "true" since timeSlot in tuesday overlaps with TDT4100Lecture in
repeatingTuesday
```

Sketch with text and code how the *delegation technique* and inheritance from **DayPlan** can be used to implement this behavior, including if and how you need to modify **DayPlan** to make the solution more tidy.

Part 4 – Input/output (IO) (10%)

a) What is the essential difference between the input/output classes **InputStream**/**OutputStream** and their sublasses compared to **Reader**/**Writer** and their subclasses?

b) A certain kind of exception is used when doing input/output, which one? In what way does this affect the code that does input/output?

c) Why do we have to close input and output "streams" with the **close()** method when we are done? How can you ensure that this happens in the case of exceptions?

NTNU
Institutt for datateknikk
og informasjonsvitenskap

Eksamensoppgåve i

TDT4100 – Objektorientert programmering

Laurdag 19. mai 2012, kl. 09:00 - 13:00

Oppgåva er utarbeidd av faglærar Hallvard Trætteberg og kvalitetssikra av Trond Aalberg.
Kontaktperson under eksamen Hallvard Trætteberg (mobil 918 97263)

Språkform: Nynorsk
Tillatne hjelpemiddel: C
Kun Java Pocket Guide, utgjeve av O'Reilly forlag, er tillaten.

Sensurfrist: Mondag 11. juni.

Les oppgåveteksten nøye. Finn ut kva det er spurt etter i kvar oppgåve.

Dersom du meiner at opplysningar manglar i ei oppgåveformulering, gjer kort greie for dei føresetnadene som du finn naudsynte.

Del 1 – Innkapsling (20%)

Gitt følgjande klasse, som representerer eit *tidspunkt* på dagen:

```
public class DayTime {

	public final int hours, minutes;

	public DayTime(int hours, int minutes) {
		this.hours = hours;
		this.minutes = minutes;
	}

	public String toString() {
		return hours + ":" + minutes;
	}
}
```

a) Kva tyder **final**-nøkkelordet slik det er brukt her? På kva slags måte(r) ivaretar denne klassen føremålet med innkapsling og på kva slags måte(r) ikkje?

b) Skildre den generelle teknikken og namnekonvensjon(ane) for å representere og kapsle inn ein enkel verdi, t.d. tal eller objektreferanse, som skal kunne endrast etter at objektet er oppretta.

c) I kva slags metoda(r) bruker ein *unnatak* ifm. innkapsling, og korleis? Vis gjerne med eit døme!

d) Mange klassar inneheld eit List<X>-felt og definerer addX, removeX og andre metodar som tilsvarer og bruker metodane i List-grensesnittet. Ei alternativ løysing kunne ha vore å arve frå ein List<X>-implementasjon, t.d. ArrayList<X>. Kvifor bruker ein aldri denne løysinga i praksis?

Del 2 – Klassar (50%)

I denne oppgåva skal du implementere klassar og metodar for å handtere ein dagsplan med tidsrom, t.d. avtalar. Klassen oppgitt i oppgåve 1 a) kan brukast som ein del av implementasjonen din. Hugs at alle metodar kan brukast i implementasjonen av andre.

a) Du skal implementere ein klasse **TimeSlot** for å representere eit *tidsrom* innanfor ein dag, t.d. ein avtale. Eit **TimeSlot**-objekt har ei skildring (tekst), eit start- og sluttidspunkt (time og minutt) og ein varigheit (minutt). Ingen av desse dataene skal kunne endrast etter at **TimeSlot**-objektet er oppretta. Du vel sjølv korleis desse dataene skal representerast og kva hjelpemetodar som evt. trengst. Merk at varigheit kan reknast ut frå starttidspunkt og sluttidspunkt, evt. at sluttidspunkt kan reknast ut frå starttidspunkt og varighet.

Implementer følgjande konstruktørar og metodar:
- **TimeSlot(String description, int hours, int minutes, int duration)**: initialiserer med oppgitt forklaring og starttidspunkt (timar og minutt) og varigheit (minutt). T.d. vil **new TimeSlot(”TDT4100-forelesning”, 10, 15, 105)** representere ei TDT4100-forelesning i tidsrommet 10:15-12:00.

- **String toString()**: returnerer ein **String** på forma **skildring@start-slutt** hvor **start** og **slutt** er på forma **tt:mm** (altså to siffer pr. tal). Dersom **toString()**-metoden blir kalla på **TimeSlot**-objektet frå førre punkt, skal det gi **"TDT4100-forelesning@10:15-12:00"**.
- **DayTime getStartTime()**: returnerer starttidspunktet som eit **DayTime**-objekt (sjå oppgåve 1).
- **DayTime getEndTime()**: returnerer sluttidspunktet som eit **DayTime**-objekt (sjå oppgåve 1).
- **int getDuration()**: returnerer varigheit i minutt.

b) Implementer følgjande metodar:
- **boolean contains(int hours, int minutes)**: returnerer om dette **TimeSlot**-objektet inneheld tidspunktet gitt med **hours** og **minutes**. Merk at sluttidspunktet ikkje blir rekna for å vere med i tidsrommet. Dette tyder at **new TimeSlot("…", 8, 0, 30).contains(8, 0)** skal gi **true**, mens **new TimeSlot("…", 8, 0, 30).contains(8, 30)** skal gi **false**.
- **boolean overlaps(TimeSlot timeSlot)**: returnerer om dette **TimeSlot**-objektet overlapper med det gitte **TimeSlot**-objektet, dvs. om det finst eit tidspunkt som begge inneheld.

c) **TimeSlot**-klassen skal støtte sortering. **TimeSlot**-objektet med tidlegast starttidspunkt blir sortert først, og dersom starttidspunkta er like, så skal det med tidlegast sluttidspunkt bli sortert først. Forklar og implementer naudsynt kode.

d) Du skal implementere ein klasse **DayPlan**, for å halde oversikt over alle avtalane (altså **TimeSlot**-objekt) for ein dag, m.a. gjere det mogeleg å leggje til og fjerne **TimeSlot**-objekt. Vel sjølv kva felt og evt. hjelpemetodar som trengst.

Implementer følgjande metodar:
- **void addTimeSlot(TimeSlot timeSlot)**: legg det gitte tidsrommet til denne dagsplanen
- **void removeTimeSlot(TimeSlot timeSlot)**: fjernar det gitte tidsrommet frå denne dagsplanen
- **TimeSlot getTimeSlotAt(int hours, int minutes)**: returnerer det *tidlegaste* tidsrommet som inneheld tidspunktet gitt med **hours** og **minutes**, ellers null.

d) Implementer følgjande to metodar for tidsplanleggjing:
- **boolean containsOverlapping()**: returnerer om det finst overlappande tidsrom i denne dagsplanen.
- **Collection<TimeSlot> getFreeTime()**: returnerer ei samling **TimeSlot**-objekt som representerer *fritida* ein har i løpet av ein dag, dvs. tidsromma som denne dagsplanen *ikkje* dekker.

For begge desse metodene kan det vere lurt å definere hjelpemetodar for å gjere løysinga ryddigare.

Del 3 – Arv og delegering (20%)

a) Du skal implementere støtte for TDT4100-forelesningar, som eit spesielt tidsrom som alltid er frå **10:15** til **12:00**. Vis korleis ein klasse **TDT4100Lecture** kan implementere dette vha. arv frå **TimeSlot**.

b) Du skal implementere støtte for ein dagsplan som alltid inneheld eit **TDT4100Lecture**-objekt, dvs. ei TDT4100-forelesning frå 10:15-12:00. Det skal ikkje vere mogeleg å fjerne TDT4100-forelesninga eller leggje inn andre tidsrom som overlappar med den. Følgjande kode illustrerer korleis det skal verke:

```
DayPlan tuesday = new TDT4100DayPlan();
System.out.println(tuesday.getTimeSlotAt(10, 30));
// prints "TDT4100 lecture@10:15-12:00"
tuesday.addTimeSlot(new TimeSlot("Coffee break", 11, 30, 60));
// throws appropriate exception, since it overlaps with the TDT4100 lecture
```

Vis korleis dette kan bli implementert i ein **TDT4100DayPlan**-klasse vha. arv frå **DayPlan**.

c) Det er ofte nyttig å kunne la éin dagsplan byggje på eller inkludere ein eller fleire andre, t.d. la dagsplanen for ein bestemt tysdag inkludere tysdagsplanen som gjeld for heile semesteret (der m.a. TDT4100-forelesninga ligg). Følgjande kode illustrerer korleis det skal verke:

```
DayPlan repeatingTuesday = new TDT4100DayPlan();
DelegatingDayPlan tuesday = new DelegatingDayPlan(repeatingTuesday);
System.out.println(tuesday.getTimeSlotAt(10, 30));
// prints "TDT4100 lecture@10:15-12:00" since tuesday logically includes the
TDT4100Lecture in repeatingTuesday

tuesday.addTimeSlot(new TimeSlot("Coffee break", 11, 30, 60));
System.out.println(tuesday.containsOverlapping());
// prints "true" since timeSlot in tuesday overlaps with TDT4100Lecture in
repeatingTuesday
```

Skisser med tekst og kode korleis *delegeringsteknikken* og arv frå **DayPlan** kan brukast for å implementere denne oppførselen, inkludert korleis du evt. vil modifisere **DayPlan** for å gjere løysinga ryddigare.

Del 4 – Input/output (IO) (10%)

a) Kva er den grunnleggjande skilnaden på input/output-klassane **InputStream**/**OutputStream** og deira subklassar ift. **Reader**/**Writer** og deira subklassar?

b) Ifm. input/output blir det brukt ein eigen type unnatak, kva slags? Og korleis påverkar dette kode som driv med input/output?

c) Kvifor må vi lukke input- og output-"straumar" med **close()**-metoden når vi er ferdige med dei? Korleis sikrar vi at det skjer også i tilfelle unnatak?