NTNU
Kunnskap for en bedre verden

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgave i TDT4100 Objektorientert programmering

Faglig kontakt under eksamen: Hallvard Trætteberg

Tlf.: 91897263

Eksamensdato: 9. august

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: 1

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. Hvis du blir bedt om å skrive kode for et eller annet formål, så definer gjerne *hjelpemetoder* dersom du synes det er hensiktsmessig.

Del 1 – Teori (20%)

Gitt følgende klasse:

```
public class Account {

	private int balance = 0;

	public int getBalance() {
		return balance;
	}

	public void deposit(int amount) {
		// ???
		balance += amount;
	}

	public void withdraw(int amount) {
		// ???
		balance -= amount;
	}
}
```

I tillegg til koden er det spesifisert at **deposit**- og **withdraw**-metodene skal utløse unntak (og ikke endre objektet) dersom beløpet som gis som argument, er negativt.

a) Hva kalles det at metoder sjekker argumentene sine før de utfører evt. endringer på objektet? Fyll inn kode for // ??? i de to metodene, slik at oppførselen blir korrekt iht. kravet gitt over.

b) Tegn et objekttilstandsdiagram for et **Account**-objekt som viser hva som skjer når sekvensen **deposit(100)**, **getBalance()**, **withdraw(150)** og **withdraw(-50)** utføres på objektet.

c) I koden over så initialiseres **balance**-feltet i deklarasjonen. Hva slags annen teknikk finnes for initialisering av et objekt og hvordan virker den? Hva er fordeler/ulemper med denne teknikken?

d) Gitt følgende kode for en **Transaction**-klasse, for å registrere data om én overføring mellom to konti:

```
public class Transaction {

	public final Account source;
	public final Account target;

	public final int amount;

	public final Date date = new Date();
}
```

Hva betyr nøkkelordet **final** her og hva mangler for å gjøre klassen korrekt gitt denne bruken av **final**? I hvilken grad støtter **Transaction**-klassen *innkapsling*?

Del 2 – Klasser (45%)

I denne oppgaven skal du implementere klasser knyttet til håndtering av konti i en bank og transaksjoner mellom dem.

Utgangspunktet for klassen for konto er gitt i oppgave 1. Du skal utvide denne til å håndtere flere krav, og implementere en klasse for banken kalt **Bank**. Oppgaven videreføres i resten av oppgavesettet, så det kan være greit å lese gjennom alle delene før du begynner å jobbe med din løsning.

a) **Kontonummer.** Du skal først utvide **Account**-klassen slik at den støtter kontonummer. Kontonummeret består av en sekvens med siffer og skal kun kunne settes ved opprettelse, og det skal kunne leses med metoden **String getAccountId()**. Skriv kode for nødvendige felt og metoder.

b) **Øvre grense for uttak.** Skriv (kode for) nødvendige felt og metoder (og evt. andre endringer i eksisterende kode) for å støtte en *øvre grense for (størrelsen til) et uttak*. Dersom beløpet som ønskes tatt ut med **withdraw**-metoden overskrider denne grensen, så skal det utløses et passende unntak.

c) **Bank-klasse med konti.** Du skal lage en **Bank**-klasse som inneholder et sett med konti (**Account**-objekter). Skriv metodene **addAccount** og **createAccount** og nødvendige felt for å støtte dem:

- **addAccount(Account account)**: registrerer en konto, dersom det ikke allerede er registrert en med samme kontonummer.
- **createAccount()**: lager et nytt **Account**-objekt med et kontonummer som ikke finnes fra før i denne banken og registrerer det. Metoden skal returnere det nye objektet.

d) **Transaksjoner**. **Bank**-klassen skal la en utføre overføringer mellom to konti, altså uttak og innskudd som hører sammen. **Transaction**-klassen fra oppgave 1 brukes for å registrere informasjon om en slik overføring. **Bank**-klassen skal ha en **transfer**-metode for å overføre et beløp mellom to registrerte konti, med følgende signatur: **void transfer(Account source, Account target, int amount)**. Oppførselen skal være som følger:

- Metoden skal utløse et unntak dersom **source**- eller **target**-kontoen ikke er registrert i denne banken.
- Metoden tar ut det angitt beløpet fra **source**-kontoen og setter det inn på **target**-kontoen.
- I tillegg *opprettes* og *lagres* (i **Bank**-objektet) et **Transaction**-objekt med data om overføringen.
- Dersom uttaket eller innskuddet *utløser unntak*, så skal **source**- og **target**-objektene *i praksis være uendret* i etterkant og transaksjonen *skal ikke lagres*. Unntaket som stoppet transaksjonen skal (gjen)utløses, slik at metoden som kaller **transfer**, kan ta det imot.

e) Implementer metoden **getTransferSum(Account account, int year, int month)**. Basert på lagrede transaksjoner, så skal den returnere summen av alle *uttak* fra **account**-argumentet som ble utført i løpet av den angitte måneden og året.

Del 3 – Input/output (IO) og unntak (10%)

I denne oppgaven skal du utvide **Bank**-klassen til å håndtere innlesing av transaksjonsdata som tekst fra fil (input-strøm). Transaksjoner skal leses linje for linje, på et format *du velger selv*. For å hjelpe deg foreslår vi følgende format: <source>**-**<target>**:**<amount>. <source> og <target> er kontonummer og <amount> er beløpet, f.eks. angir **1-2:150** en overføring på 150 fra konto 1 til konto 2.

Implementer følgende metode for å *lese inn* transaksjonsdata og *utføre* transaksjonene:

- **doTransactions(InputStream input)** – metoden skal lese inn transaksjoner fra input-strømmen på formatet du har valgt selv (evt. som angitt over) og *utføre* transaksjonen. Dersom en transaksjon har feil format eller ikke kan utføres, så skal den stilltiende ignoreres. Du kan anta at **Bank**-objektet som metoden utføres på, allerede er konfigurert med et sett konti.

Del 4 – Arv (15%)

I denne delen skal du bruke *arv* og lage en subklasse av **Bank** kalt **SafeBank**, som gir muligheten for å angi månedlige uttaksgrenser (for summen av uttak) pr. konto i banken. Implementasjonen skal gjøre det mulig å registrere ulike grenser (eller ingen) for hver konto, *uten* at (koden for) **Account**-klassen endres. Dersom **transfer**-metoden er i ferd med å foreta et uttak slik at grensen (for summen av uttak) overstiges, så skal transaksjonen avbrytes og et passende unntak utløses.

Forklar med tekst og kode hvordan du vil implementere **SafeBank**, inkludert hvordan du vil endre **Bank** for å gjøre løsningen ryddigst mulig. Men merk at selv om du endrer **Bank**, så skal oppførselen være den samme, f.eks. skal gamle metoder ikke ha noen annen synlig effekt enn før og evt. nye metoder skal ikke la andre klasser kunne endre tilstanden på ulovlig vis.

Del 5 – Testing (10%)

Skriv testkode for å sjekke at **Account** oppfører seg som beskrevet i oppgave 1 (og dokumentert i objekttilstandsdiagrammet ditt). Du kan, men trenger ikke bruke JUnit-rammeverket, siden det viktigste er den generelle testmetoden.

NTNU – Trondheim
Norwegian University of
Science and Technology

Department of computer and information science

Examination paper for TDT4100
Object-oriented programming with Java

Academic contact during examination: Hallvard Trætteberg

Phone: 91897263

Examination date: 9. August

Examination time (from-to): 9:00-13:00

Permitted examination support material: C

Only ”Big Java”, by Cay S. Horstmann, is allowed.

Other information:

This examination paper is written by teacher Hallvard Trætteberg, with quality assurance by Ragnhild Kobro Runde (Ifi, UiO).

Language: English

Number of pages: 4

Number of pages enclosed: 1

Checked by:

Date Signature

If you feel necessary information is missing, state the assumptions you find it necessary to make. If you are not able to *implement* classes and method that a part asks for, you may still use these classes and methods later. If you are asked to write code for some purpose, feel free to define utility/helper methods to improve the code.

Part 1 – Theory (20%)

Given the following class:

```
public class Account {

	private int balance = 0;

	public int getBalance() {
		return balance;
	}

	public void deposit(int amount) {
		// ???
		balance += amount;
	}

	public void withdraw(int amount) {
		// ???
		balance -= amount;
	}
}
```

In addition to this code, it is specified that the **deposit** and **withdraw** methods must throw an exception (and not modify the object) if the amount argument is negative.

a) What do you call it when methods check their arguments before modifying the object? Fill in code for // ??? in the two methods, so the behaviour is in accordance with the requirement stated above.

b) Draw an object state diagram for an **Account** object that shows what happens when the sequence **deposit(100)**, **getBalance()**, **withdraw(150)** and **withdraw(-50)** is performed on the object.

c) In the code above, the **balance** field is initialized in the declaration. What other mechanism for initializing objects exists and how does it work? What are advantages/disadvantages with this mechanism?

d) Given the following code for a **Transaction** class, for registering data about a single transfer between two accounts:

```
public class Transaction {

	public final Account source;
	public final Account target;

	public final int amount;

	public final Date date = new Date();
}
```

What does the keyword **final** mean here and what is missing for making the class correct given this usage of **final**? To what extent does the **Transaction** class support *encapsulation*?

Part 2 – Classes (45%)

In this part you will implement classes for managing accounts in a bank and transaction between them.

The starting point for the code for an account class is provided in part 1. You must extend it to handle further requirements and also write code for a **Bank** class. The code will be developed further in later parts, so it is wise to read through all parts before designing your solution.

a) **Account number.** You must first extend the **Account** class so it supports an account number. An account number is a sequence of digits and should only by possible to set when the account is instantiated, and it should be possible to read with the method **String getAccountId()**. Write code for necessary fields and methods.

b) **Upper limit for withdrawals.** Write (code for) necessary fields and methods (and other changes in existing code you find necessary) to support an *upper limit for (the size of) a withdrawal*. If the amount provided to the **withdraw** method is above this limit, a suitable exception should be thrown.

c) **Bank class with accounts.** You must write a **Bank** class that contains a set of accounts (**Account** objects). Write the methods **addAccount** and **createAccount** and necessary supporting fields:

- **addAccount(Account account)**: registers an account, if no account with the same account number is registered.
- **createAccount()**: creates a new **Account** object with an account number that is not yet registered in this bank, and registers this **Account** object. The method must return the new object.

d) **Transactions.** The **Bank** class must support transferring amounts between two accounts, i.e. pairs of withdrawals and deposits. The **Transaction** class from part 1 is used for registering information about such a transfer. The **Bank** class must have a **transfer** method for transferring an amount between two registered accounts, with the following signature: **void transfer(Account source, Account target, int amount)**. Its behaviour must be as follows:

- The method must throw an exception if the **source** or **target** accounts are not registered in this bank.
- The method must withdraw the provided amount from the **source** account and deposit it in the **target** account.
- In addition, a **Transaction** object must be *created* and *stored* (in this **Bank** object) with data about the transfer.
- If the withdrawal or deposit *throws an exception*, the **source** and **target** objects must *remain unmodified after the call* and the transaction must *not be stored*. The exception that aborted the transaction must be (re)thrown, so the method that called **transfer**, can catch it.

e) Implement the method **getTransferSum(Account account, int year, int month)**. Based on the stored transactions, it must return the sum of all *witdrawals* from the **account** argument that was performed during the provided month and year.

Part 3 – Input/output (IO) and exceptions (10%)

In this part you must extend the **Bank** class to handle reading of transactions as text from a file (input stream). The transactions must be read one line at a time, in a format *you chose yourself*. As help, we suggest the following format: <source>**-**<target>**:**<amount>. <source> and <target> are account numbers and <amount> is the amount, e.g. **1-2:150** represents a transfer of 150 from account 1 to account 2.

Implement the following method for *reading in* transaction data and *performing* the transactions:

- **doTransactions(InputStream input)** – the method must read transactions from the input stream in the chosen format (or as suggested above) and *perform* the transactions. If a transaction has the wrong format or cannot be performed, it must be silently ignored. You can assume that the **Bank** object for which the method is called, already is configured with a set of accounts.

Part 4 – Inheritance (15%)

In this part you must use *inheritance* and make a subclass of **Bank** named **SafeBank**, that supports setting monthly limits for (the sum of) withdrawals for each account in the bank. The implementation must support setting different limits (or none) for each account, *without* changing the (code of the) **Account** class. If the **transfer** method is about to perform a withdrawal that will exceed the limit (for the sum of withdrawals), then the transaction must be aborted and a suitable exception thrown.

Explain with text and code how you would implement **SafeBank**, including how you will modify **Bank** to the make the solution as tidy as possible. But not that if you modify **Bank**, its behaviour must remain the same, e.g. old methods must not have a different visible effect and any new methods cannot allow other classes to modify the state in illegal ways.

Part 5 – Testing (10%)

Write test code for checking that **Account** behaves as specified in part 1 (and documented in your object state diagram). You can, but don't need to, use the JUnit framework, since it is the general testing method that is important.

NTNU
Kunnskap for ei betre verd

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgåve i TDT4100
Objektorientert programmering

Fagleg kontakt under eksamen: Hallvard Trætteberg

Tlf.: 91897263

Eksamensdato: 9. august

Eksamenstid (frå-til): 9.00-13.00

Hjelpemiddelkode/Tillatne hjelpemiddel: C

Berre ”Big Java”, av Cay S. Horstmann, er tillaten.

Annan informasjon:

Oppgåva er utarbeidd av faglærer Hallvard Trætteberg og kvalitetssikra av Ragnhild Kobro Runde (Ifi, UiO).

Målform/språk: Nynorsk

Sidetal: 4

Sidetal vedlegg: 1

Kontrollert av:

_____

Dato Sign

Om du meiner at opplysningar manglar i ein oppgåveformulering, gjer kort greie for dei antakingar og føresetnader som du finn naudsynt. Om du i ein del er beden om å *implementere* klasser og metodar og du ikkje klarar det (heilt eller delvis), så kan du likevel *nytte* dei i seinare delar. Om du blir beden om å skrive kode for eit eller anna føremål, så definer gjerne *hjelpemetodar* om du synst det er hensiktsmessig.

Del 1 – Teori (20%)

Gitt fylgjande klasse:

```
public class Account {

	private int balance = 0;

	public int getBalance() {
		return balance;
	}

	public void deposit(int amount) {
		// ???
		balance += amount;
	}

	public void withdraw(int amount) {
		// ???
		balance -= amount;
	}
}
```

I tillegg til koden er det spesifisert at **deposit**- og **withdraw**-metodane skal utløyse unntak (og ikkje endre objektet) om beløpet som er gjeven som argument, er negativt.

a) Kva kallast det at metodar sjekkar argumenta sine før dei utfører evt. endringar på objektet? Fyll inn kode for // ??? i de to metodane, slik at oppførselen blir korrekt iht. kravet gitt over.

b) Teikn eit objekttilstandsdiagram for eit **Account**-objekt som viser kva som skjer når sekvensen **deposit(100)**, **getBalance()**, **withdraw(150)** og **withdraw(-50)** utførast på objektet.

c) I koden over så initialiserast **balance**-feltet i deklarasjonen. Kva for anna teknikk finst for initialisering av eit objekt og korleis fungerer den? Kva er fordeler/ulemper med denne teknikken?

d) Gitt fylgjande kode for ein **Transaction**-klasse, for å registrere data om ein overføring mellom to konti:

```
public class Transaction {

	public final Account source;
	public final Account target;

	public final int amount;

	public final Date date = new Date();
}
```

Kva tyder nøkkelordet **final** her og kva manglar for å gjere klassen korrekt gitt denne bruken av **final**? I kva for grad støtter **Transaction**-klassen *innkapsling*?

Del 2 – Klasser (45%)

I denne oppgåva skal du implementere klasser knytt til handtering av konti i en bank og transaksjonar mellom dei.

Utgangspunktet for klassen for konto er gjeven i oppgåve 1. Du skal utvide denne til å handtere fleire krav, og implementere ein klasse for banken kalla **Bank**. Oppgåva førast vidare i resten av oppgåvesettet, så det kan være greitt å lese gjennom alle delane før du tar til å jobbe med di løysing.

a) **Kontonummer.** Du skal først utvide **Account**-klassen slik at den støtter kontonummer. Kontonummeret består av ein sekvens med siffer og skal berre kunne settast ved oppretting, og det skal kunne lesast med metoden **String getAccountId()**. Skriv kode for naudsynte felt og metodar.

b) **Øvre grense for uttak.** Skriv (kode for) naudsynte felt og metodar (og evt. andre endringar i eksisterande kode) for å støtte ein *øvre grense for (storleiken til) eit uttak*. Om beløpet som ynskjast tatt ut med **withdraw**-metoden overskrid denne grensa, så skal det løysast ut eit passande unntak.

c) **Bank-klasse med konti.** Du skal lage ein **Bank**-klasse som inneheld eit sett med konti (**Account**-objekt). Skriv metodane **addAccount** og **createAccount** og naudsynte felt for å støtte dei:

- **addAccount(Account account)**: registrerar en konto, dersom det ikkje er registrert ein med same kontonummer frå før.
- **createAccount()**: lager eit nytt **Account**-objekt med eit kontonummer som ikkje finst frå før i denne banken og registrerar det. Metoden skal returnere det nye objektet.

d) **Transaksjonar**. **Bank**-klassen skal la ein utføre overføringar mellom to konti, altså uttak og innskot som høyrer saman. **Transaction**-klassen frå oppgåve 1 nyttast for å registrere informasjon om ein slik overføring. **Bank**-klassen skal ha ein **transfer**-metode for å overføre eit beløp mellom to registrerte konti, med fylgjande signatur: **void transfer(Account source, Account target, int amount)**. Oppførselen skal vere som fylgjar:

- Metoden skal løyse ut eit unntak om **source**- eller **target**-kontoen ikkje er registrert i denne banken.
- Metoden tar ut det angjevne beløpet fra **source**-kontoen og set det inn på **target**-kontoen.
- I tillegg *opprettast* og *lagrast* (i **Bank**-objektet) eit **Transaction**-objekt med data om overføringa.
- Om uttaket eller innskotet *løysar ut unntak*, så skal **source**- og **target**-objekta *i praksis vere uendra* i etterkant og transaksjonen *skal ikkje lagrast*. Unntaket som stoppa transaksjonen skal løysast ut (igjen), slik at metoden som kalla **transfer**, kan ta det imot.

e) Implementer metoden **getTransferSum(Account account, int year, int month)**. Basert på lagra transaksjonar, så skal den returnere summen av alle *uttak* frå **account**-argumentet som vart utført i løpet av den angjevne månaden og året.

Del 3 – Input/output (IO) og unntak (10%)

I denne oppgåva skal du utvide **Bank**-klassen til å handtere lesing av transaksjonsdata som tekst frå fil (input-straum). Transaksjonar skal lesast linje for linje, på et format *du velgjer sjølv*. For å hjelpe deg foreslår vi fylgjande format: <source>**-**<target>**:**<amount>. <source> og <target> er kontonummer og <amount> er beløpet, f.eks. angir **1-2:150** en overføring på 150 frå konto 1 til konto 2.

Implementer fylgjande metode for å *lese inn* transaksjonsdata og *utføre* transaksjonane:

- **doTransactions(InputStream input)** – metoden skal lese inn transaksjonar frå input-straumen på formatet du har vald sjølv (evt. som angjeven over) og *utføre* transaksjonen. Dersom ein transaksjon har feil format eller ikkje kan utførast, så skal den stilltiande ignorerast. Du kan anta at **Bank**-objektet som metoden utførast på, er konfigurert med eit sett konti frå før.

Del 4 – Arv (15%)

I denne delen skal du nytta *arv* og lage en subklasse av **Bank** kalla **SafeBank**, som gir moglegheiten for å angi månadlege uttaksgrenser (for summen av uttak) pr. konto i banken. Implementasjonen skal gjere det mogleg å registrere ulike grensar (eller inga) for kvar konto, *utan* at (koden for) **Account**-klassen endrast. Om **transfer**-metoden er i ferd med å foreta eit uttak slik at grensa (for summen av uttak) overstigast, så skal transaksjonen brytast av og et passende unntak løysast ut.

Forklar med tekst og kode korleis du vil implementere **SafeBank**, inkludert korleis du vil endre **Bank** for å gjere løysninga ryddigast mogleg. Men merk at sjølv om du endrar **Bank**, så skal oppførselen vere den same, f.eks. skal gamle metodar ikkje ha nokon annan synleg effekt enn før og evt. nye metodar skal ikkje la andre klasser kunne endre tilstanden på ulovleg vis.

Del 5 – Testing (10%)

Skriv testkode for å sjekke at **Account** oppfører seg som skriven i oppgåve 1 (og dokumentert i objekttilstandsdiagrammet ditt). Du kan, men treng ikkje nytte JUnit-rammeverket, sidan det viktigaste er den generelle testmetoden.

Appendix

Fra java.util.Date-klassen:

java.util.Date.Date()

Allocates a `Date` object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.

int java.util.Date.getYear()

Returns a value that is the result of subtracting 1900 from the year that contains or begins with the instant in time represented by this `Date` object.

int java.util.Date.getMonth()

Returns a number representing the month that contains or begins with the instant in time represented by this `Date` object. The value returned is between `0` and `11`, with the value `0` representing January.

int java.util.Date.getDay()

Returns the day of the week represented by this date. The returned value (`0` = Sunday, `1` = Monday, ... `6` = Saturday) represents the day of the week that contains or begins with the instant in time represented by this `Date` object.