NTNU
Kunnskap for en bedre verden

Institutt for datateknikk og informasjonsvitenskap

# Kontinuasjonseksamensoppgave 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.

Dette kalles *validering* (og er en viktig del av innkapsling). Det er lurt å skille ut valideringen i en egen metode som kalles fra endringsmetodene (her **deposit** og **withdraw**). Denne kan gjerne være **protected**, så subklasser kan redefinere og gjenbruke valideringslogikken. Det er vanligst å bruke en såkalt *unchecked exception* (sjekket unntak), som **IllegalArgumentException**.

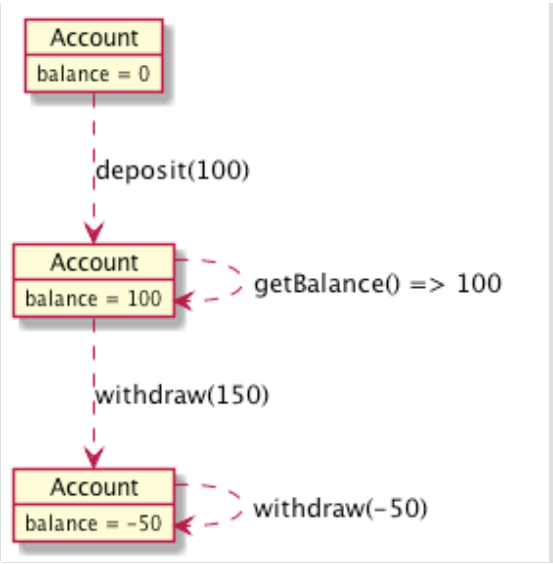
```
protected void checkAmount(int amount) {
	if (amount < 0) {
		throw new IllegalArgumentException("Amount cannot be negative, but was " +
amount);
	}
}

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

public void withdraw(int amount) {
	checkAmount(amount);
```

```
        balance -= amount;
}
```

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.




Her er oppgaven dels å forstå koden, dels å kjenne diagram-notasjonen, spesielt at en transisjon (metodekall) leder tilbake til samme tilstand, dersom metodekallet ikke endrer 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?

En annen teknikk for initialisering er å bruke en eller flere konstruktører. Dette er metoder med samme navn som klassen som blir implisitt kalt ved bruk av new (med eller uten argumenter).
Fordeler:

- en kan skrive mer komplisert kode for initialisering, med f.eks. valg og validering
- en kan (tvinges til å) oppgi argumenter som kan brukes i initialiseringen
- konstruktører hjelper en å sikre tilstand ved bruk av innkapsling

Ulemper:

- koden for initialisering blir ofte langt unna feltene den initialiserer
- det er lett å glemme å initialisere felt

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*?

**final** betyr her at feltet ikke skal kunne endres etter at det er initialisert. Da må en enten initialisere i deklarasjonen eller i en eller flere konstruktører, eller en kombinasjon. Her er det naturlig å innføre en konstruktør som setter de tre feltene som ikke er initialisert i deklarasjonen:

```
public Transaction(Account source, Account target, int amount) {
	this.source = source;
	this.target = target;
	this.amount = amount;
}
```

Innkapsling handler om å 1) hindre at en kan sette objekter til en ulovlig tilstand og 2) skjule representasjonsdetaljer slik at en kan gjøre endringer uten at andre klasser også må endres. Her er innkapsling delvis støttet, siden det ikke er mulig å gjøre tilstanden gal. Imidlertid så skjules ikke hvordan data er representert, siden feltene er deklarert som **public**.

## 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.

```
private final String accountId;

private void checkAccountId(String accountId) {
	for (int i = 0; i < accountId.length(); i++) {
		char c = accountId.charAt(i);
		if (! Character.isDigit(c)) {
			throw new IllegalArgumentException(c + " is not a valid account id
character");
		}
	}
}

public Account(String accountId) {
	checkAccountId(accountId);
	this.accountId = accountId;
}

public String getAccountId() {
	return accountId;
}
```

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.

En må innføre et **int**-felt med (i hvert fall) en set-metode og endre withdraw-metoden så den sjekker beløpet opp mot dette feltet.

```
private int withdrawLimit = 0;

public void setWithdrawLimit(int withdrawLimit) {
	this.withdrawLimit = withdrawLimit;
}

protected void checkWithdrawAmount(int amount) {
	if (withdrawLimit > 0 && amount > withdrawLimit) {
		throw new IllegalArgumentException("Withdraw amount cannot be above " +
withdrawLimit + ", but was " + amount);
	}
}

public void withdraw(int amount) {
	checkAmount(amount);
	checkWithdrawAmount(amount);
	balance -= amount;
}
```

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.

```
private List<Account> accounts = new ArrayList<Account>();

protected Account getAccount(String accountId) {
	for (Account account : accounts) {
		if (accountId.equals(account.getAccountId())) {
			return account;
		}
	}
	return null;
}

protected boolean exists(String accountId) {
	return getAccount(accountId) != null;
}

public void addAccount(Account account) {
	if (! exists(account.getAccountId())) {
		accounts.add(account);
	}
}
```

```
private int nextAccountId = 1;

public Account createAccount() {
	while (exists(String.valueOf(nextAccountId))) {
		nextAccountId++;
	}
	Account account = new Account(String.valueOf(nextAccountId));
	addAccount(account);
	return account;
}
```

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.

```
protected List<Transaction> transactions = new ArrayList<Transaction>();

protected void checkTransaction(Transaction transaction) {
}

public void transfer(Account source, Account target, int amount) throws Exception {
	if (! (accounts.contains(source) && accounts.contains(target))) {
		throw new IllegalArgumentException("A bank can only handle registered
accounts");
	}
	Transaction transaction = new Transaction(source, target, amount);
	checkTransaction(transaction);
	source.withdraw(amount);
	try {
		target.deposit(amount);
	}
	catch (Exception e) {
		source.deposit(amount);
		throw e;
	}
	transactions.add(transaction);
}
```

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.

```
private int getTransferSum(Account account, int year, int month) {
```

```
        int sum = 0;
        for (Transaction transaction : transactions) {
                if (transaction.source == account) {
                        Date date = transaction.date;
                        if (date.getYear() == year && date.getMonth() == month) {
                                sum += transaction.amount;
                        }
                }
        }
        return sum;
}
```

## 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.

```
public void doTransactions(InputStream input) {
        Scanner scanner = new Scanner(input);
        while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                int dashPos = line.indexOf('-'), colonPos = line.indexOf(':');
                try {
                        Account source = getAccount(line.substring(0, dashPos));
                        Account target = getAccount(line.substring(dashPos + 1, colonPos));
                        int amount = Integer.valueOf(line.substring(colonPos + 1));
                        if (source == null || target == null) {
                                continue;
                        }
                        transfer(source, target, amount);
                } catch (Exception e) {
                }
        }
        scanner.close();
}
```

## 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 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 endre tilstanden på ulovlig vis.

```
private Map<Account, Integer> monthlyTransferLimits = new HashMap<Account, Integer>();

public void setMonthlyTransferLimit(Account account, int limit) {
	monthlyTransferLimits.put(account, limit);
}

@Override
protected void checkTransaction(Transaction transaction) {
	Account source = transaction.source;
	int amount = transaction.amount;
	Date date = transaction.date;
	Integer transferLimit = monthlyTransferLimits.get(source);
	if (transferLimit != null) {
		int transferSum = getTransferSum(source, date.getYear(), date.getMonth());
		if (transferSum + amount > transferLimit) {
			throw new IllegalStateException("Monthly transfer limit exceeded");
		}
	}
}
```

## 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.

```
public void testObjectStateDiagram() {
	assertEquals(0, account.getBalance());
	account.deposit(100);
	assertEquals(100, account.getBalance());
	assertEquals(100, account.getBalance());
	account.withdraw(150);
	assertEquals(-50, account.getBalance());
	try {
		account.withdraw(-50);
	} catch (Exception e) {
	}
	assertEquals(-50, account.getBalance());
}
```

# 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.