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: 6. juni
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:

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.

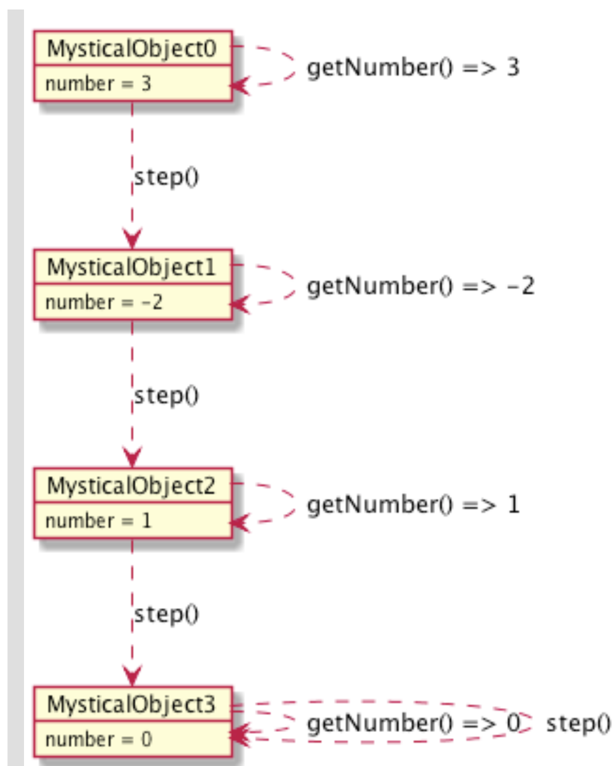
Del 1 – Teori (20%)

Gitt følgende klasse:

```
public class MysticalObject {

	private int number;

	public MysticalObject(int number) {
		this.number = number;
	}

	public void step() {
		number = -(number - (int) Math.signum(number));
	}

	public int getNumber() {
		return number;
	}
}
```

Se vedlegget for dokumentasjon til **Math.signum**-metoden.

a) Tegn et objekttilstandsdiagram for et **MysticalObject**-objekt som er instansiert med **new MysticalObject(3)**, som viser oppførselen til **getNumber**- og **step**-metodene.



Her er oppgaven dels å forstå koden, dels å kjenne diagramnotasjonen og dels å skjønne hva som er relevant å ha med.

b) Objekttilstandsdiagrammer kan brukes til å beskrive oppførselen til et objekt. Angi en viktig fordel og en ulempe/begrensning ved slik bruk.

Fordeler: Eksempel på forløp kan være enklere å forstå enn en komplett definisjon i form av regler (invarianter). Diagrammer kan være mer intuitive enn tekst. Det er lett å skrive test-kode basert på diagrammet.
Ulemper: Diagrammene blir lett store, hvis de skal dekke alle relevante tilfeller. Det er kun i enkle tilfeller en kan beskrive oppførselen komplett.

c) Java krever at en deklarerer *typen* til alle felt, variabler og parametre (i motsetning f.eks. Python, Javascript og Matlab). Hva er de viktigste fordelene dette gir?

De blir lettere å:

- oppdage/hindre feil bruk av verdier (for verktøy, kompilator og programmerer)
- tilby hjelp til kodingen, f.eks. foreslå metoder
- kompilere til effektiv kode

d) I koden over så står det **(int)** foran kallet til **Math.signum**. Hva kalles dette (denne typen *uttrykk*) og hvordan virker det (påvirker det utførelsen) i dette tilfellet? Den samme syntaksen kan også brukes i tilfeller hvor typen er en klasse eller et grensesnitt. Hvordan virker det da?

Det kalles **casting** og brukes for å "tvinge" Java til å "akseptere" at uttrykket som helhet har den angitte typen. Når det brukes på tall-typer, så vil Java *konvertere* fra den ene til den andre typen (her: fra **float** til **int**). Når casting brukes med klasser/grensesnitt så vil det *sjekkes* at det indre uttrykket har den angitt typen, og hvis ikke så utløses **ClassCastException**.

Del 2 – Klasser (30%)

I denne oppgaven skal du implementere klasser og metoder for en quiz, altså sekvenser med spørsmål og svar. Et eksempel på spørsmål og svar-sekvens er vist under. Som eksemplet viser, så kan det kan være ulike typer svar og en kan ha svar-alternativer.

```
Hva heter hovedstaden i Norge?
Oslo
Hva slags ost er månen laget av? 1. Camembert 2. Roquefort 3. Brie
2
Hvor høy er Galdhøpiggen? 1. 2469 2. 2471
2
Feil, prøv igjen!
2469
Er Java gøy?
ja
```

Du skal implementere én klasse for spørsmål og tilhørende svar kalt **Question** og én klasse for quiz-en kalt **Quiz**. 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) Du skal først implementere **Question**-klassen. Klassen skal støtte spørsmål, svar og eventuelle svar-alternativer, som alle er tekster. Definer felt for disse og skriv en (eller flere) konstruktør(er) som initialiserer dem. Merk at et spørsmål *må* ha et svar, men *trenger ikke* ha svar-alternativer. Utløs unntak for tilfeller som du synes er relevante.

String-typen brukes for tekst uten formattering. Siden det er flere svar-alternativer, så lagrer vi dem i en **ArrayList<String>**. Siden vi siden trenger å kunne slå opp svar-alternativer med *indeks*, så deklareres feltet som **List<String>**. Vi viser her et alternativ med to konstruktører, den andre med varargs, men det holder med én.

```
	private String question;
	private String answer;
	private List<String> options;

	public Question(String question, String answer, Collection<String> options) {
		this.question = question;
		this.answer = answer;
		if (! options.contains(answer)) {
			throw new IllegalArgumentException("Svaret er ikke et av alternativene!");
		}
		this.options = new ArrayList<String>();
		if (options != null) {
			this.options.addAll(options);
		}
	}

	public Question(String question, String answer, String... options) {
		this(question, answer, Arrays.asList(options));
	}
```

b) Lag følgende metoder i **Question**-klassen for å stille spørsmål og sjekke svar:

- **askQuestion(PrintStream)**: stiller spørsmålet ved å skrive tekst til **PrintStream**-argumentet. Eventuelle svar-alternativer skal også skrives ut. Nummerer alternativene fra 1 og utover, så det er lettere å skrive inn nummeret som svar.
- **checkAnswer(String answer)**: sjekker svaret angitt som argument og returnerer **true** om det er riktig, og **false** ellers. Dersom spørsmålet har svar-alternativer, så skal det både være lov å skrive svaret og angi *nummeret på svar-alternativet* (fortsatt som en **String**). Ta høyde for tilfellet hvor et spørsmål har et tall som riktig svar og har svar-alternativer, slik som i Galdhøpiggen-spørsmålet i eksemplet.

Her får en vist at en kan enkel gjennomgang av liste. Det er viktig å få med seg poenget med at nummereringen starter på 1, mens indekser (om de brukes) i lista starter på 0.

NB: Her var det dessverre mange som ble forvirret av at det ikke var oppgitt noe parameternavn bak PrintStream-typen og heller ikke skjønte at PrintStream fungerte som System.out (som er en PrintStream). Likevel, de burde skjønt at de kunne bruke print/println iht. vedlagte dokumentasjon.

```
	public void askQuestion(PrintStream out) {
```

```
        out.print(question);
        if (options.size() > 0) {
            out.println("Alternativer:");
            int num = 1;
            for (String option : options) {
                out.println(num + ". " + option);
                num++;
            }
        }
    }
```

Her er det viktig at en både får sjekket svaret som indeks i evt. svar-alternativer og som direkte svar. Dette krever riktig kontrollstruktur, inkludert håndtering av evt. unntak. Det er riktignok litt fiklete, så det trekkes ikke mye om det ikke blir helt riktig.

```
    public boolean checkAnswer(String answer) {
        if (options.size() > 0) {
            try {
                int num = Integer.valueOf(answer);
                if (this.answer.equals(options.get(num - 1))) {
                    return true;
                }
            } catch (IndexOutOfBoundsException e) {
            } catch (NumberFormatException e) {
            }
        }
        return this.answer.equals(answer);
    }
```

c) Det er ofte et skjønnspørsmål om en trenger get- og set-metoder og evt. andre metoder for å lese og endre tilstanden i et objekt. Begrunn hvilke slike metoder som trengs (om noen) i **Question**-klassen og implementer dem.

Generelt er det en fordel å redusere settet med metoder, siden det gir frihet til å endre klasse siden. Når felt initaliseres i en konstruktør, så trengs som oftest ikke set-metoder. Her trenger en heller ikke get-metoder, siden metodene som stiller spørsmål og sjekker svar utgjør et komplett API for klassen.

d) **Quiz**-klassen representerer en sekvens med spørsmål. Skriv kode for klassen med metoder for å:

- legge til **Question**-objekter
- stille spørsmål med utskrift til skjerm (**System.out**) og lese inn svar fra tastaturet (**System.in**) helt til det er svart riktig på hvert spørsmål, omtrent som vist i eksemplet i innledningen.

Siden det var mange som ikke skjønte (visste) at System.out er en PrintStream, så må en nesten godta litt kreative svar ifm. kallet til askQuestion i run()-metoden.

```
    private Collection<Question> questions = new ArrayList<Question>();

    public void addQuestion(Question question) {
        questions.add(question);
    }
```

```java
	public void run() {
		Scanner scanner = new Scanner(System.in);
		for (Question question : questions) {
			question.askQuestion(System.out);
			while (scanner.hasNextLine()) {
				String answer = scanner.nextLine();
				if (question.checkAnswer(answer)) {
					break;
				}
				System.out.println("Feil, prøv igjen!");
				question.askQuestion(System.out);
			}
		}
		scanner.close();
	}
```

e) Skriv nødvendige metoder for å kunne *kjøre* **Quiz**-klassen (med noen eksempelspørsmål) som et hovedprogram iht. reglene som Java har for dette. Når man kjører **Quiz**-klassen, skal altså brukeren bli stilt spørsmålene og få anledning til å svare, omtrent som vist i eksemplet i innledningen.

```java
	public void init() {
		addQuestion(new Question("Hva heter hovedstaden i Norge?", "Oslo"));
		addQuestion(new Question("Hva slags ost er månen laget av?",
"Roquefort", "Camembert", "Roquefort", "Brie"));
		addQuestion(new Question("Hvor høy er Galdhøpiggen?", "2469", "2469",
"2471"));
		addQuestion(new Question("Er Java gøy?", "ja"));
	}

	public static void main(String[] args) {
		Quiz quiz = new Quiz();
		quiz.init();
		quiz.run();
	}
```

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

Utvid **Quiz**-klassen med en metode for å initialisere **Quiz**-objektet med spørsmål (og tilhørende svar og evt. svar-alternativ) fra fil. Følgende format skal støttes:

- Spørsmål, svar og evt. svar-alternativer har én linje hver.
- Spørsmålet kommer først, så svaret og deretter evt. svar-alternativer.
- Svar-alternativene skilles fra neste spørsmål med en tom linje.

Eksempelfil tilvarende eksemplet i innledningen, med kommentarer (som ikke er en del av filinnholdet) i høyre kolonne:

Fil	Kommentar
Hva heter hovedstaden i Norge?	1. spørsmål
Oslo	riktig svar
	skillelinje
Hva slags ost er månen laget av?	2. spørsmål

Roquefort	riktig svar
Camembert	1. svar-alternativ
Roquefort	2. svar-alternativ
Brie	3. svar-alternativ
	skillelinje
Hvor høy er Galdhøpiggen?	3. spørsmål
2469	riktig svar
2469	1. svar-alternativ
2471	2. svar-alternativ
	skillelinje
Er Java gøy?	4. spørsmål
ja	riktig svar

```java
	public void init(Reader input) throws IOException {
		BufferedReader reader = new BufferedReader(input);
		while (reader.ready()) {
			String question = reader.readLine();
			if (question == null || question.trim().length() == 0) {
				break;
			}
			String answer = reader.readLine();
			Collection<String> options = new ArrayList<String>();
			while (reader.ready()) {
				String line = reader.readLine();
				if (line == null || line.trim().length() == 0) {
					break;
				}
				options.add(line);
			}
			addQuestion(new Question(question, answer, options));
		}
	}
```

Del 4 – Arv (15%)

I denne delen skal du bruke *arv* for å håndtere ulike typer svar på en ryddigere måte. Det skal være én klasse for frie tekst-svar kalt **StringQuestion**, en klasse for tekst-svar med svar-alternativer kalt **StringOptionsQuestion**, og en klasse for ja/nei-spørsmål kalt **BooleanQuestion**. Det som er felles for disse klassene, f.eks. håndtering av selve spørsmålsteksten, skal samles i **Question**-superklassen. Bortsett fra at **Question**-klassen ikke skal kunne instansieres, så skal *bruken* av den være som i del 3, inkludert de to **askQuestion**- og **checkQuestion**-metodene fra del 2. Prøv å strukturere klassene dine, så det blir *minst mulig duplisert kode*. Du står selvsagt fritt til å definere andre metoder som trengs i løsningen din.

En del av koden vil være lik tidligere kode. Du kan selv velge om du vil skrive den på nytt, eller beskrive presist hvordan tidligere skrevet kode(tekst) kopieres inn i de nye klassene.

a) Implementer først **Question**-superklassen og de tre klassene **StringQuestion**, **StringOptionsQuestion** og **BooleanQuestion** kun med konstruktører.

Her er det viktig å identifisere at **Question**-klassen bør være abstrakt og deklarere både **askQuestion**- og **checkAnswer** (abstrakt). **Question**-klassen bør ha en konstruktør for å sette spørsmålet og implementere **askQuestion**. Subklassene må ta inn spørsmålet i sine konstruktører og bruke super(…) for å sette spørsmålet. Koden forøvrig bør være omtrent den samme som i del 2.

```
public abstract class Question {

	private String question;

	protected Question(String question) {
		this.question = question;
	}

	public void askQuestion(PrintStream out) {
		out.println(question);
	}

	public abstract boolean checkAnswer(String answer);
}

public class StringQuestion extends Question {

	private String answer;

	public StringQuestion(String question, String answer) {
		super(question);
		this.answer = answer;
	}

	@Override
	public boolean checkAnswer(String answer) {
		return this.answer.equals(answer);
	}
}
```

Vi har her valgt å arve fra StringQuestion, for å gjenbruke mest mulig logikk.

```
public class StringOptionsQuestion extends StringQuestion {

	private List<String> options;

	public StringOptionsQuestion(String question, String answer,
Collection<String> options) {
		super(question, answer);
		this.options = new ArrayList<String>(options);
		if (! options.contains(answer)) {
			throw new IllegalArgumentException("Svaret er ikke et av
alternativene!");
		}
		this.options = new ArrayList<String>(options);
	}
```

```
	@Override
	public void askQuestion(PrintStream out) {
		super.askQuestion(out);
		int num = 1;
		for (String option : options) {
			out.println(num + ". " + option);
			num++;
		}
	}

	@Override
	public boolean checkAnswer(String answer) {
		try {
			int num = Integer.valueOf(answer);
			if (this.answer.equals(options.get(num - 1))) {
				return true;
			}
		} catch (IndexOutOfBoundsException e) {
		} catch (NumberFormatException e) {
		}
		return super.checkAnswer(answer);
	}
}
```

Denne klassen bør passe på å bruke **boolean** for å lagre svaret.

```
public class BooleanQuestion extends Question {

	private boolean answer;

	public BooleanQuestion(String question, boolean answer) {
		super(question);
		this.answer = answer;
	}

	@Override
	public boolean checkAnswer(String answer) {
		return (this.answer ? "ja" : "nei").equals(answer);
	}
}
```

b) Implementer de to metodene **askQuestion** og **checkAnswer**, slik at alle **Question**-objekter (egentlig instanser av en av de tre andre klassene) i praksis virker som i del 2.

Løsningen er vist over.

c) Reimplementer metoden i **Quiz** for for innlesing av spørsmål fra fil, slik at den virker med de nye **Question**-subklassene.

Her er det stor sett samme kode som i del 3, men merk at en må instansiere de ulike subklassene til **Question** og vite når de ulike variantene skal brukes. Det er greit å referere til tidligere skrevet kode,

for å spare tid og plass.

```
	public void init(Reader input) throws IOException {
		BufferedReader reader = new BufferedReader(input);
		while (reader.ready()) {
			String question = reader.readLine();
			if (question == null || question.trim().length() == 0) {
				break;
			}
			String answer = reader.readLine();
			Collection<String> options = new ArrayList<String>();
			while (reader.ready()) {
				String line = reader.readLine();
				if (line == null || line.trim().length() == 0) {
					break;
				}
				options.add(line);
			}
			if (answer.equals("ja") && options.size() == 0) {
				addQuestion(new BooleanQuestion(question, true));
			} else if (answer.equals("nei")") && options.size() == 0) {
				addQuestion(new BooleanQuestion(question, false));
			} else if (options.size() > 0) {
				addQuestion(new StringOptionsQuestion(question, answer, 
options));
			} else {
				addQuestion(new StringQuestion(question, answer));
			}
		}
	}
```

Del 5 – Trinnvis utførelse (15%)

I denne delen skal **Quiz**-klassen endres slik at den holder rede på tilstanden til en runde med spørsmål og svar, men lar en *annen* (hovedprogram)klasse styre fremdriften. Følgende tre metoder skal brukes til å styre fremdriften:

- **start(boolean mode, PrintStream out, InputStream in)**: starter quiz-en (men ingen spørsmål stilles ennå). **mode**-argumentet angir om et galt besvart spørsmål gjentas *med en gang* (**mode**=**false**), eller *etter at* alle etterfølgende spørsmål er stilt (**mode**=**true**). **out**-argumentet er strømmen som spørsmål skal skrives til (gjøres i **doQuestion**). **in**-argumentet er strømmen som svarene skal leses fra (gjøres også i **doQuestion**).
- **doQuestion()**: Stiller ett (neste) spørsmål og leser ett svar(forsøk). Hvilket spørsmål som stilles avgjøres av hvilke som ennå ikke er stilt og besvart riktig og **mode**-verdien som ble gitt til **start**-metoden. Metoden returnerer antall spørsmål som *ennå ikke er riktig besvart*.
- **stop()**: stopper quiz-en og returnerer hvor mange spørsmål som *ble riktig besvart*.

Følgende kode eksemplifiserer hvordan (denne versjonen av) **Quiz**-klassen er ment å bli brukt:

```
	Quiz quiz = new Quiz();
	// initialiser fra fil her (ikke vist)
	quiz.start(true, System.out, System.in); // start quiz
```

```
    while (quiz.doQuestion() > 0); // still spørsmål så lenge flere gjenstår
    quiz.stop();
```

Implementer **start**-, **doQuestion**- og **stop**-metodene.

Her er det viktig å skjønne at en må lagre argumentene til **start**-metoden, så en kan bruke dem siden, og skjønne hvordan en skal representere og håndtere gjenstående spørsmål iht. **mode**-verdien.

Her tillater vi litt mer skissemessig kode, siden det sentrale er å skjønne hva som må huskes på tvers av kallene til start og **doQuestion**.

```
    private boolean mode;
    private PrintStream out;
    private Scanner scanner;
    private int correctCount;
    private List<Question> remaining;

    public void start(boolean mode, PrintStream out, InputStream in) {
        this.out = out;
        scanner = new Scanner(in);
        correctCount = 0;
        remaining = new ArrayList<Question>(questions);
    }

    public int doQuestion() {
        Question question = remaining.remove(0);
        question.askQuestion(out);
        String answer = scanner.nextLine();
        if (question.checkAnswer(answer)) {
            System.out.println("Riktig");
            correctCount++;
        } else {
            System.out.println("Feil");
            remaining.add(mode ? remaining.size() : 0, question);
        }
        return (remaining != null ? remaining.size() : -1);
    }

    public int stop() {
        scanner.close();
        return correctCount;
    }
```

<u>Del 6 – Grensesnitt (10%)</u>

Det er ønskelig å kunne støtte ulike filformat for quiz-er. Forklar med tekst og kode hvordan du vil bruke Java-grensesnitt for å gjøre det enkelt å bytte mellom ulike format og hvordan filinnlesingskoden du allerede har skrevet kan utgjøre (implementasjonen av) standardformatet. Merk at du ikke skal implementere nye format, kun vise hvordan det lett kan gjøres.

```
public interface QuizFormat {
```

```java
	public Collection<Question> read(Reader input) throws IOException;
}

public class StandardQuizFormat implements QuizFormat {

	public Collection<Question> read(Reader input) throws IOException {
		Collection<Question> questions = new ArrayList<Question>();
		//
		// kode som i init-metoden,
		// men nye spørsmål legges til den interne questions-lista
		//
		return questions;
	}
}
```

I Quiz:

```java
	private QuizFormat quizFormat = new StandardQuizFormat();

	public void setQuizFormat(QuizFormat quizFormat) {
		this.quizFormat = quizFormat;
	}

	public void init(Reader input) throws IOException {
		questions.addAll(quizFormat.read(input));
	}
```

Appendix

Fra Math-klassen:

`float java.lang.Math.signum(float f)`

Returns the signum function of the argument; zero if the argument is zero, 1.0f if the argument is greater than zero, -1.0f if the argument is less than zero.

Fra PrintStream-klassen:

`java.io.PrintStream`

A PrintStream adds functionality to another output stream, namely the ability to print representations of various data values conveniently.

`void java.io.PrintStream.print(String s)`

Prints a string. If the argument is null then the string "null" is printed. Otherwise, the string's characters are converted into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write(int) method.

`void java.io.PrintStream.println(String x)`

Prints a String and then terminate the line. This method behaves as though it invokes print(String) and then println().