NTNU
Kunnskap for en bedre verden

Institutt for datateknikk og informasjonsvitenskap

# Kontinuasjonseksamensoppgave i TDT4100 Objektorientert programmering

**Faglig kontakt under eksamen: Hallvard Trætteberg**

**Tlf.: 918 97 263**

**Eksamensdag: Fredag 19. 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 Rune Sætre.

**Målform/språk: Bokmål**

**Antall sider: 3**

**Antall sider vedlegg: 6**

**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.

En oversikt over klasser og metoder for alle oppgavene er gitt i vedlegg 3. Kommentarene inneholder krav til de ulike delene, som du må ta hensyn til når du løser oppgavene. I tillegg til metodene som er oppgitt, står du fritt til å definere ekstra metoder for å gjøre løsningen ryddigere. Nyttige standardklasser og -metoder finnes i vedlegg 4.

## Del 1 – Teori (15%)

I vedlegg 1 finner du en datamodell som viser de tre klassene **Person**, **Course** og **Exam**.

a) Hva betyr de ulike diagramdelene, altså boksene, strekene/pilene og tegnene (ord, tall og *)?

Diagrammet er omtrent som et klassediagram, så boksene tilsvarer klasser. Navnet øverst en boks er klassenavnet, mens de andre er egenskaper, dvs. data, som instansene vil ha. Strekene er assosiasjoner (eller relasjoner), som sier noe om hvordan instanser kan kobles sammen. Tallene på enden av strekene angir hvor mange koblinger en instans kan ha (* betyr ubegrenset), såkalt *multiplisitet*. Pilene sier noe om i hvilken retning en kan følge en kobling, ingen betyr begge retninger, mens én pil betyr bare i den retningen.

b) Hvilke valg må en typisk ta når en skal skrive Java-kode for klassene basert slike diagrammer?

Egenskapene og assosiasjonene blir typisk til felt med type som passer til multiplisiteten. En må velge hvordan dataene skal *kapsles inn*, altså konstruktører med parametre, gettere og settere, og evt. add- og remove-metoder og andre metoder.

c) **Exam**-klassen bruker en **char**-verdi for å representere en eksamenskarakter. Hva er problematisk med dette med tanke på å sikre korrekt bruk av klassen? Skisser kort, med tekst og/eller kode, en alternativ teknikk som løser problemet.

En kan hindre gal bruk, dvs. ulovlige char-verdier, med validering og unntak, men det er bedre å kunne sjekke slikt ved kompilering. Da trengs en egen **Grade**-klasse og denne kan kodes slik at det ikke finnes andre instanser enn de som tilsvarer de lovlig karakterene A-F. Dette gjøres ved å ha privat(e) konstruktør(er) og én statisk variabel for hver mulig verdi:

```
private Grade(char grade) { ... }
public static Grade A = new Grade('A'), B = new Grade('B'), ... , F = new Grade('F');
```

Nå kan en bare bruke lovlige instanser når det forventes et **Grade**-objekt. Dette er essensielt en manuell koding av en enum-klasse, som ikke er pensum.

## Del 2 – Course- og Exam-klassene (30%)

**Course**-klassen tilsvarer et undervist emne i et bestemt semester. Kurskoden (**code**) oppgis ved opprettelsen av **Course**-objekter, mens studiepoeng (**credits**) og semesteret (**time**) kan settes senere.

a) Deklarer felt for **time**-egenskapen, og skriv kode for **getYear**, **getSemester**, **getTime** og **setTime** (se vedlegg 3). Med tanke på del 4, så kan det være lurt å lage en metode som sjekker formatet til **setTime** sitt **time**-argument, som kan brukes i andre klasser.

Her velger vi å lagre året og semesteret separat, og la getTime og setTime gi en illusjon om et time-felt. **checkTime**-metoden er laget for også å bli brukt ved innlesing av time-data.

```
	private int year;
	private char semester; // F(all) eller S(pring)

	public int getYear() {
		return year;
	}

	public char getSemester() {
		return semester;
	}

	public String getTime() {
		return String.valueOf(getSemester()) + year;
	}

	public static void checkTime(String time) throws IllegalArgumentException {
		char c0 = Character.toUpperCase(time.charAt(0));
		if (c0 != 'S' && c0 != 'F') {
			throw new IllegalArgumentException("Semester must be either F(all) or
S(pring)");
		}
		Integer.valueOf(time.substring(1));
	}

	public void setTime(String time) {
		checkTime(time);
		int year = Integer.valueOf(time.substring(1));
		if (year < 100) {
			year = (year < 50 ? 2000 : 1900) + year;
		}
		this.semester = Character.toUpperCase(time.charAt(0));
		this.year = year;
	}
```

b) **Course**-objekter skal kunne sorteres kronologisk på semesteret de undervises. Forklar hvilke endringer som må gjøres på **Course**-klassen for å kunne bruke Java sin standard mekanisme for sortering, og skriv den nødvendige koden.

Dersom **Course**-klassen implementerer **Comparable**, dvs. sammenligning med et annet **Course**-objekt, så kan **Collections.sort**- og **List.sort**-metodene brukes til sortering av **Course**-objekter.

```
public class Course implements Comparable<Course> {

	// Compares based on the time given, earlier means smaller.
	@Override
	public int compareTo(Course other) {
		if (year != other.year) {
			return year - other.year;
		}
		return (other.semester - semester);
	}
}
```

**Exam**-klassen tilsvarer en avlagt eksamen for et undervist emne. Både emnet (**course**) og karakteren (**grade**) oppgis ved opprettelse av **Exam**-objekter.

c) Deklarer felt for **course**- og **grade**-egenskapene, og skriv konstruktøren og **isPass**-metoden (se vedlegg 3).

```
	private final Course course;
	private char grade;

	public Exam(Course course, char grade) {
		this.course = course;
		grade = Character.toUpperCase(grade);
		if ("ABCDEF".indexOf(grade) < 0) {
			throw new IllegalArgumentException("Result must be one of the
characters A-F");
		}
		this.grade = grade;
	}

	public boolean isPass() {
		return grade != 'F';
	}
```

d) **Exam**-objekter skal også kunne sorteres, men på to måter! Standard-sorteringen skal være kronologisk på semesteret kurset ble undervist, men **Exam**-objektene skal også kunne sorteres på karakter. Forklar hvordan begge sorteringene kan støttes og skriv nødvendig kode.

I tillegg til **Comparable**, som bygger sorteringsrekkefølgen inn i klassen selv, så kan en implementere en **Comparator<Exam>**, som er en annen klasse som sammenligne to **Exam**-objekter:

```
public class Exam implements Comparable<Exam> {

	@Override
	public int compareTo(Exam other) {
		return course.compareTo(other.getCourse());
	}
}

public class ExamComparator implements Comparator<Exam> {

	@Override
	public int compare(Exam exam1, Exam exam2) {
		return exam1.getGrade() - exam2.getGrade();
	}
}
```

## Del 3 – Person-klassen (20%)

**Person**-klassen er knyttet til **Course**- og **Exam**-klassene, og tanken er at **Course**-objekter legges til når en person melder seg opp og **Exam**-objekter legges til når eksamen er avlagt. En skal bare kunne ta eksamen i kurs en har meldt seg opp i!

a) Skriv kode for **name**-egenskapen, gitt at den bare skal kunne settes ved opprettelse av **Person**-objekter.

```
	private final String name;

	public Person(String name) {
		this.name = name;
	}

	public String getName() {
		return name;
	}
```

b) Deklarer felt for **courses**-egenskapen, og begrunn valg av type. Skriv kode for **addCourse** og **hasCourse** (se vedlegg 3).

Typen bør være et grensesnitt fra Collection-rammeverket, enten **Collection** eller **List**, som er spesialisert til element-typen **Course**. Her brukes **Collection**, fordi en ikke trenger andre metoder enn den deklarerer.

```
	private Collection<Course> courses = new ArrayList<Course>();

	public boolean addCourse(Course course) {
		for (Course existing : courses) {
			if (course.getTime().equals(existing.getTime())) {
				return false;
			}
		}
		courses.add(course);
		return true;
	}

	public boolean hasCourse(String code) {
		for (Course course : courses) {
			if (code.equals(course.getCode())) {
				return true;
			}
		}
		return false;
	}
```

c) Deklarer felt for **exams**-egenskapen, og begrunn valg av type. Skriv kode for **addExam**, **getLastExam** og **hasPassed** (se vedlegg 3).

Samme begrunnelse for typen som over.

```
	private Collection<Exam> exams = new ArrayList<Exam>();

	public Exam addExam(Course course, char grade) {
		if (courses.contains(course)) {
			for (Exam exam : exams) {
				if (exam.getCourse() == course && exam.isPass()) {
					return null;
				}
			}
			Exam exam = new Exam(course, grade);
			exams.add(exam);
			return exam;
```

```
        }
        return null;
    }

    public Exam getLastExam(String code) {
        Exam lastExam = null;
        for (Exam exam : exams) {
            if (code.equals(exam.getCourse().getCode())) {
                lastExam = exam;
            }
        }
        return lastExam;
    }

    public boolean hasPassed(String code) {
        Exam exam = getLastExam(code);
        return exam != null && exam.isPass();
    }
```

d) Skriv kode for **countCredits**-metoden i **Person**-klassen. Hust at en ikke får studiepoeng for kurs uten å ha tatt og bestått eksamen, og at det er siste karakter som teller!

```
    public double countCredits() {
        // first collect all course codes
        Collection<String> codes = new ArrayList<String>();
        for (Exam exam : exams) {
            String code = exam.getCourse().getCode();
            if (! codes.contains(code)) {
                codes.add(code);
            }
        }
        // check last exam result for each course code and add credit if passed
        double sum = 0;
        for (String code : codes) {
            Exam exam = getLastExam(code);
            if (exam.isPass()) {
                sum += exam.getCourse().getCredits();
            }
        }
        return sum;
    }
```

## Del 4 – IO (10%)

Vedlegg 3 beskriver et tekstformat for informasjon om emner og eksamener.

a) Skriv metoden **Collection<Exam> readExams(Reader input)** i en tenkt **ExamReader**-klasse, som skal opprette **Course**- og **Exam**-objekter tilsvarende teksten lest fra **input**-argumentet, og returnere alle **Exam**-objektene. Det kreves ikke spesifikk håndtering av feil format ut over at metoden ikke skal utløse unntak.

```
    public Collection<Exam> readExams(Reader input) {
        Collection<Exam> exams = new ArrayList<>();
        Scanner scanner = new Scanner(input);
        String lastSemester = null;
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
```

```
                String[] tokens = line.split(" ");
                if (tokens.length == 1) {
                        try {
                                Course.checkTime(tokens[0]);
                                lastSemester = tokens[0];
                                continue;
                        } catch (IllegalArgumentException e) {
                        }
                }
                try {
                        Course course = new Course(tokens[0]);
                        course.setTime(lastSemester);
                        course.setCredits(Double.valueOf(tokens[1]));
                        for (int i = 2; i < tokens.length; i++) {
                                exams.add(new Exam(course, tokens[i].charAt(0)));
                        }
                } catch (RuntimeException e) {
                }
        }
        scanner.close();
        return exams;
    }
```

b) Tegn et objektdiagram for objektene som opprettes ved innlesing av eksempelet i vedlegg 2.

En må ha med to **Course**-objekter, med samme emnekode (**"TDT4100"**), semesterbokstav (**'S'**) og studiepoeng (**7,5**), men forskjellig år (**2016** og **2017**). Så har en to **Exam**-objekter koblet til det første, med karakterene **'F'** og **'C'**, og ett **Exam**-objekt med karakteren **'A'** koblet til den andre. Merk at det er viktigst at diagrammet er konsistent med koden, så det kan være annerledes enn beskrevet her.

## Del 5 – ExamRequirement-klassen og IExamRequirement-grensesnittet (25%)

**ExamRequirement**-klassen (se vedlegg 3) representerer en sjekk for om et **Exam**-objekt, altså en avlagt eksamen, tilfredsstiller visse krav. F.eks. kan en lage et **ExamRequirement**-objekt som sjekker om en har fått minst C i TDT4100. Selve testen gjøres av **accepts**-metoden, som sjekker det angitte **Exam**-argumentet mot verdiene som ligger i **ExamRequirement**-objektet selv.

a) Skriv ferdig konstruktør nr. 2, og gjør den så kort og enkel som mulig.

```
    public ExamRequirement(String course, int sinceYear) {
        this(course, sinceYear, 'E');
    }
```

b) **accepts**-metoden m/hjelpemetoden **acceptsCourse** er ferdigskrevet, men koden inneholder (minst) tre feil. Skriv korrekt kode.

```
    private boolean acceptsCourse(Course course) {
        return (! this.course.equals(course.getCode())) && course.getYear() <
sinceYear;
    }
```

c) Nederst i klassen er feltet **atLeastCInTdt4100** deklarert. Med koden **atLeastCInTdt4100.accepts(**...**)** skal en kunne sjekke om en har minst C i TDT4100. Skriv ferdig deklarasjonen, både modifikator(er) og initialiseringsuttrykket.

```
	public final static ExamRequirement atLeastCInJava = new ExamRequirement("TDT4100",
0, 'C');
```

d) Du trenger å sjekke om en eksamen er for et masteremne på IDI, som har kode som begynner på ”TDT42”, f.eks. TDT4250. Forklar hvordan du kan bruke arv for å organisere koden og hvordan eksisterende kode evt. må endres.

I **acceptsCourse** så sammenlignes navn med **equals**, mens vi trenger å sammenligne med **startsWith**. En kan lage en **protected**-metode kalt **acceptsCourseCode** i **ExamRequirement**, som kalles av **acceptsCourse** og som brukes **equals**. Så redefineres denne metoden i en subklasse, hvor en bruker **startsWith**.

I ExamRequirements:

```
	private boolean acceptsCourse(Course course) {
		return (! acceptsCourseCode(course.getCode())) && course.getYear() <
sinceYear;
	}

	protected boolean acceptsCourseCode(String course) {
		return course.equals(this.course);
	}
```

I subklassen:

```
	@Override
	protected boolean acceptsCourseCode(String course) {
		return course.startsWith(this.course);
	}
```

e) Skriv kode for et *funksjonelt* grensesnitt **IExamRequirement**, som **ExamRequirement** (allerede implisitt) implementerer. Forklar hvorfor et slikt grensesnitt kan være nyttig, fremfor å bare ha **ExamRequirement**-klassen.

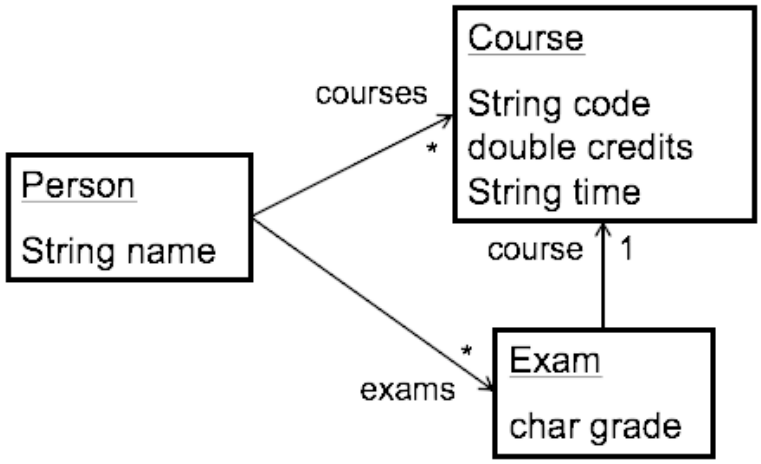
Men funksjonelt menes et grensesnitt med bare én metode (og denne metoden gir samme resultat for om den kalles flere ganger på samme exam-objekt). Her er det bare én mulig metode i **ExamRequirement** som er aktuell som grensesnitt-metoden. Bruk av et slikt grensesnitt gir større frihet i hvilke krav en kan implementere og hvordan det gjøres.

```
public interface IExamRequirement {
	public boolean accepts(Exam exam);
}
```

f) Skriv en *alternativ* deklarasjon av **atLeastCInTdt4100** som har **IExamRequirement** som type og utnytter Java 8 sin funksjonssyntaks i initialiseringsuttrykket.

```
	public static IExamRequirement atLeastCInJave =
			(exam) -> "TDT4100".equals(exam.getCourse().getCode()) &&
exam.getGrade() >= 'C';
```

## Appendix 1: Class overview




## Appendix 2: File format for exam results

The file format is line-based and has two kinds of lines:

1) *Semester lines* contain only one item, which is a valid time of the format used by **Course**'s **setTime** method. A semester line may be followed by zero, one or more *course lines*.
2) *Course lines* contain a course code followed by a credit value and at least one exam grade for that course. A single space is used as separator. All the courses are taught in the semester indicated in the previous *semester line*.

Example of the text format. The left column shows the text contents, and the right column explains the meaning:

| | |
|---|---|
| `S2016`<br>`TDT4100 7.5 F C`<br>`S2017`<br>`TDT4100 7.5 A` | Semester line for the Spring 2016 semester.<br>Course line for TDT4100, with grades F and C.<br>Semester line for the Spring 2017 semester.<br>Course line for TDT4100, with grade A! |

## Appendix 3: Provided code (fragments)

```
public class Course {

      ... fields, constructors and methods for code and credits ...

      public int getYear() {  ... }

      public char getSemester() { ... }

      /**
       * Gets the time this Course is given, in the format <semester><year>
       * E.g. if the semester is 'S' and the year is 2016,
       * it should return S2016.
       */
      public String getTime() { ... }

      /**
       * Sets the time that this Course is taught. The format is the semester
       * (char) followed by the year. The year may be shortened to two digits;
       * if it is below 50 then 2000 should be added, otherwise 1900.
       * E.g. S16 means Spring 2016, while F86 means Fall 1986.
       * @param time The time in the format <semester><year>
       * @throws IllegalArgumentException if the format is incorrect
       */
      public void setTime(String time) {
            ...
      }
}

public class Exam {

      ... fields and methods for course and grade ...

      /**
       * Initialises an Exam, by setting the course and grade.
       * The grade can only be set to one of the characters 'A'-'F'.
       * @throws IllegalArgumentException if the grade is not legal
       */
      public Exam(...) { ... }

      /**
       * Tells whether this Exam has a result that is a passing grade.
       */
      public boolean isPass() { ... }
}
```

```java
public class Person {

    ... fields, constructors and methods for name ...

    /**
     * Adds a Course to this Person, if no Course is registered
     * for the same code, year and semester.
     * @param course
     * @return true if the course was added, false otherwise
     */
    public boolean addCourse(Course course) { ... }

    /**
     * Returns whether this Person has a Course with the given code.
     * @param code
     */
    public boolean hasCourse(String code) { ... }

    /**
     * Creates and adds an exam to this Person for the provided Course and
     * with the provided grade, but only if this Person has this Course and
     * no passing Exam is registered for that Course.
     * @param course
     * @param grade
     * @return the newly created and added Exam, or null
     */
    public Exam addExam(Course course, char grade) { ... }

    /**
     * Gets the exam that was registered last for the provided course code.
     * This is the exam that counts for that course!
     * @param course
     */
    public Exam getLastExam(String code) { ... }

    /**
     * Returns true of this Person has passed the Course for the provided code.
     * @param code
     */
    public boolean hasPassed(String code) { ... }

    /**
     * Counts the credits this Person has earned.
     */
    public double countCredits() { ... }
}
```

```java
/**
 * Represents a requirement concerning an Exam for a specific Course.
 * The Exam's result must not be before the provided year (sinceYear) and
 * the grade must not be worse than the provided grade (minGrade).
 */
public class ExamRequirement {

	public final String course;
	public final int sinceYear;
	public final char minGrade;

	public ExamRequirement(String course, int sinceYear, char minGrade) {
		this.course = course;
		this.sinceYear = sinceYear;
		this.minGrade = minGrade;
	}

	/**
	 * Initialises with the course and year with provided data and
	 * makes sure a passing grade is required.
	 * @param course
	 * @param sinceYear
	 */
	public ExamRequirement(String course, int sinceYear) { ... }

	/**
	 * Helper method for checking the course
	 * @param course
	 */
	private boolean acceptsCourse(Course course) {
		return this.course == course && sinceYear > course.getYear();
	}

	/**
	 * Returns true if the provided Exam is for the specified course,
	 * not before the specified year and not worse than the specified grade.
	 * @param exam
	 * @return
	 */
	public boolean accepts(Exam exam) {
		return acceptsCourse(exam.getCourse()) && exam.getGrade() >=
minGrade;
	}

	// declares atLeastCInTdt4100
	... ExamRequirement atLeastCInTdt4100 = ...;
}
```