NTNU
Kunnskap for en bedre verden

Institutt for datateknikk og informasjonsvitenskap

Eksamensoppgave i TDT4100 Objektorientert programmering

Faglig kontakt under eksamen: Hallvard Trætteberg

Tlf.: 918 97 263

Eksamensdag: Lørdag 11. 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: 8 (5+1+2)

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 1. 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 3.

Del 1 – Gender- og Person-klassene (30%)

Gender-klassen (vedlegg 1) representerer kjønnet til en person, i denne oppgaven begrenset til mann og kvinne. Klassen skal kodes slik at det ikke skal være mulig å ha andre **Gender**-objekter enn disse to kjønnene. En **Gender**-instans har et forklarende ord/navn knyttet til seg, som settes når instansen lages, og som ikke skal være direkte tilgjengelig fra andre klasser.

a) Hvilke *modifikatorer* bør stå foran deklarasjonen av **label**-feltet og konstruktøren? Begrunn svaret.

Begge disse bør være markert som **private**, siden de ikke skal være tilgjengelig utenfor klassen. **label** bør også være **final**, siden den ikke skal kunne endres.

b) Hvordan er klassen kodet så det forklarende ordet/navnet vises ved utskrift, f.eks. med **System.out.println(...)**?

Ved utskrift så brukes implisitt **toString()**-metoden, som er implementert og returnerer **label**.

c) Fullfør **valueOf**-metoden.

Her er det viktig å returnere et eksisterende Gender-objekt, ved å sammenligne argumentet med label-verdien i de to konstantene MALE og FEMALE. En kan også bruke switch, som i Java 8 virker med String-objekter. **valueOf** ligner på **Integer.valueOf** og **Double.valueOf** og må være **static**, siden det er unaturlig og unødvendig å kalle den på en eksisterende instans.

```
public static Gender valueOf(String label) {
    if (MALE.label.equals(label)) {
          return MALE;
    } else if (FEMALE.label.equals(label)) {
          return FEMALE;
    }
    return null;
}
```

Person-klassen (vedlegg 1) representerer en person, med et navn (**String**) og et kjønn (**Gender**). Navnet settes kun ved opprettelsen, mens kjønnet kan settes når som helst.

For å håndtere familieforhold så har et **Person**-objekt data om mor, far og barn. **addChild**-metoden brukes for å knytte et barn til en forelder, og det er ikke spesifisert andre metoder som endrer barn-forelder-koblingen. **addChild** både registrerer barnet og setter barnets kobling til mor eller far,

avhengig av kjønnet til forelderen. Ta f.eks. kallet **chris.addChild(pat)**, hvor **pat** blir registert som barnet til **chris**. Hvis **chris** er *mann*, så blir han registrert som **pat** sin *far*, mens hvis **chris** er *kvinne*, så blir hun registrert som **pat** sin *mor*.

d) Skriv felt, metoder og konstruktør for innkapsling av navn, kjønn, mor, og far. Bruk **name**, **gender**, **mother** og **father** som grunnlag for navngiving.

Her er litt av poenget at en må ha en konstruktør som setter navnet og utelate set-metoder for name, mother og father.

```
private String name;
private Gender gender = null;
private Person father, mother;

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

public String getName() {
    return name;
}

public Gender getGender() {
    return gender;
}

public void setGender(Gender gender) {
    this.gender = gender;
}

public Person getMother() {
    return mother;
}

public Person getFather() {
    return father;
}
```

e) Implementer **getChildCount**, **hasChild** og **getChildren**, med nødvendig(e) felt.

Her må en ha riktig deklarasjon av feltet (helst **Collection,** sekundært **List**), og huske å initialisere feltet. **getChildren** må passe på å returnere en ny **Collection**, så annen kode ikke får tilgang til interne data. En må ta høyde for at **gender** kan være **null**. En kan gjerne bruke **Stream**-teknikken.

```
private Collection<Person> children = new ArrayList<>();

public int getChildCount() {
    return children.size();
}

public boolean hasChild(Person child) {
```

```
    return children.contains(child);
}

public Collection<Person> getChildren(Gender gender) {
    Collection<Person> result = new ArrayList<>();
    for (Person child : children) {
          if (gender == null || child.getGender() == gender) {
                result.add(child);
          }
    }
    return result;
}
```

f) Som en del av innkapslingen av barn-koblingen, så har vi valgt å la **Person** implementere **Iterable<Person>**-grensesnittet. Hva betyr dette for koden en kan skrive for å gå gjennom barna til en **Person**? Implementer metoden(e) som er påkrevd av dette grensesnittet.

Hvis en klasse implementerer **Iterable**, så kan referanser til denne klassen brukes på høyresiden av kolonet i en for-each-løkke, f.eks. for (Person child : person). Se også **getChildren**-koden over. Et (litt mindre relevant, og ikke påkrevd) alternativ er **Iterable.forEach(Comsumer<Person>)**. Dette er en såkalt **default**-metode som en får gratis når en implementerer **Iterable**.

```
@Override
public Iterator<Person> iterator() {
    return children.iterator();
}
```

g) Implementer **addChild**-metoden, basert på kravene beskrevet over og i vedlegg 1, og kravene som implisitt testes av **testAddChild** i **PersonTest**-klassen (vedlegg 1).

```
public void addChild(Person child) {
    if (getGender() == Gender.MALE) {
          if (child.father != null) {
                child.father.children.remove(child);
          }
          child.father = this;
    } else if (getGender() == Gender.FEMALE) {
          if (child.mother != null) {
                child.mother.children.remove(child);
          }
          child.mother = this;
    }
    children.add(child);
}
```

h) Nederst i testmetoden **testAddChild** er to linjer marker med // **???** Hvilke tilfeller eller problemer er det som kan testes på de to punktene i koden? Hvordan oppfører din kode seg, og hva mener du er riktig oppførsel i de to tilfellene? (Du trenger ikke rette på koden din)

Her er det vanskelig å komme med en fasit, men poenget er å vurdere oppførselen til egen kode for tilfeller en ikke har tenkt på. En bør kunne beskrive de to tilfellene og forstå hva som faktisk vil skje i egen kode.

I det første tilfellet skifter far først kjønn, før hun på ny får lagt til barnet. Koden over vil fjerne koblingen til og fra den eksisterende moren, og så etablere en tilsvarende kobling med den nye moren, som også er den eksisterende faren. Hun ender opp med å være far og mor til samme barn, og dette barnet vil finnes dobbelt opp i **children**-lista. Her ville nok det riktigste være at den eksisterende far-koblingen også ble fjernet, før mor-barn-koblingen ble etablert.
I det andre tilfellet så legges faren til som barn av eget barn, som gir en sirkulær struktur. Dette bør det være en sjekk for, for mange algoritmer vil da ende opp i evig løkke.

Del 2 – Family-klassen og IO (25%)

Family-klassen (vedlegg 1) holder rede på personer i en familie, med metoder for å legge til familiemedlemmer (**Person**-objekter), slå opp personer på navn, lagre familiemedlemmene og lese dem inn igjen.

a) Skriv metodene **addMember** og **findMember** (og definer nødvendige felt), som henholdsvis legger en **Person** til som familiemedlem og finner et familiemedlem med et angitt navn.

Samme krav til **members**-lista (navnet er ikke nøye) som til **Person.children**. En trenger ikke sørge for at alle barn av personer som legges til, også legges til. Dette kan en anta gjøres utenfra. Må bruke **equals** for å sammenligne **String**-objekter.

```
private Collection<Person> members = new ArrayList<>();

public void addMember(Person person) {
    members.add(person);
}

public Person findMember(String name) {
    for (Person person : members) {
          if (person.getName().equals(name)) {
                return person;
          }
    }
    return null;
}
```

I vedlegg 2 beskrives et tekstformat for data om personene i en familie, inkludert foreldre-barn-koblingene.

b) I vedlegg 1 er det med en hjelpemetode **tokenize**, som kan være nyttig ved innlesing og som kan antas ferdig implementert. Hvilke(n) modifikator(er) burde den ha? Begrunn svaret!

En slik hjelpemetode bør for det første være markert som **private**, siden det ikke er naturlig at dette er en tjeneste som tilbys andre klasser. For det andre bør den være markert som **static**, siden den ikke bruker (leser eller endrer) tilstanden til noe **Family**-objekt. Eneste grunn til at den ikke

skal være **static**, er hvis en subklasse av Family har behov for å redefinere den, og det er ikke aktuelt her.

c) Skriv metodene **save** og **load**, som støtter dette tekstformatet. Begrunn hvordan du velger å behandle problemet med unntak.

Det viktigste med **save**-metoden er at den først skriver ut alle linjer av type 1, altså den person-informasjonen som er nødvendig for å **lage** Person-objektene før foreldre-barn-koblingen etableres. Vi velger å lage en **PrintWriter** rundt **OutputStream**-en vi får inn, for å muliggjøre bruke av **print** og **println**. Vi kunne brukt en **PrintStream**, men en **Writer** anbefales jo for tekst (trekker ikke for bruk av **PrintStream**). Så skrives alle linjene av type 2 ut. Derfor blir det to iterasjoner over alle medlemmene. Navn får anførselstegn (") rundt (merk måten " inkluderes i en String). Her sjekkes det om en person har barn (kan gjøre på mange måter), så det ikke blir linjer med en forelder, men det er strengt tatt ikke definert som et krav (det står "sequence of names", og en sekvens kan jo ha bare ett element). Hvis en har linjer med bare én forelder, så er det viktig at load-metoden håndterer det riktig. Det er vanlig at den som setter opp en **OutputStream** også lukker den, og derfor avslutter vi *ikke* med **pw.close()**. Vi avslutter imidlertid med **pw.flush()** for å sikre at all vår output sendes ut med en gang (trekker ikke for manglende bruk av **close()**/**flush()**).

load-metoden klassifiserer hver linje som en av de tre typene ved å først sjekke om den er tom eller starter med # (type 3) og så sjekker om første token i en linje er en gyldig **Gender** (type 1). Ellers er den av type 2. Her gjøres det ingen sjekk på om formatet er korrekt, f.eks. om et barn i en linje av type 2 faktisk er registrert som familiemedlem. Det er kanskje litt uklart hvorvidt og evt. hvordan **tokenize** håndterer #, så det er greit at den brukes før en sjekker for linjer av type 3.

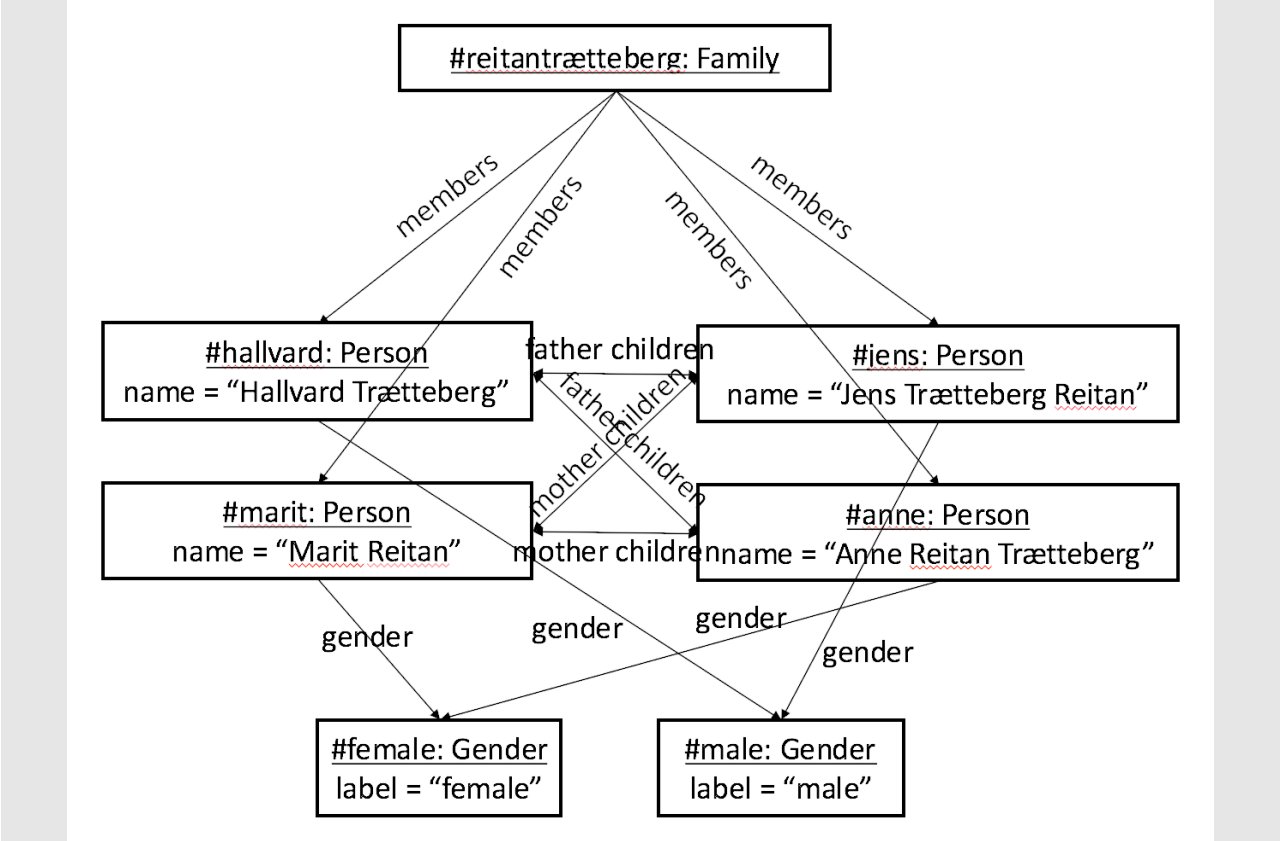
Unntak håndteres ikke av metodene, så de må deklareres med **throws**. Det er naturlig å bruke **IOException**, for den utløses ved bruk av **InputStream** og **OutputStream**. En kunne brukt **Exception** or å markere (at vi er klar over) at det er mye som kan gå galt, men det anbefales å bruke den mest spesifikke typen. Vi kunne fanget opp og ignorert unntak, men det kan lett maskere feil vi ønsker å avdekke.

```
private void outputQuotedName(Person person, PrintWriter pw) {
    pw.print("\"" + person.getName() + "\"");
}

public void save(OutputStream out) throws IOException {
    PrintWriter pw = new PrintWriter(out);
    pw.println("# all persons");
    for (Person person : members) {
          pw.print(person.getGender());
          pw.print(" ");
          outputQuotedName(person, pw);
          pw.println();
    }
    pw.println();
    pw.println("# all mother/father-child relations");
    for (Person person : members) {
          if (person.iterator().hasNext()) {
                outputQuotedName(person, pw);
                for (Person child : person) {
                      pw.print(" ");
```

```
                        outputQuotedName(child, pw);
                }
                pw.println();
        }
    }
    pw.flush();
}

public void load(InputStream in) throws IOException {
    Scanner scanner = new Scanner(in);
    while (scanner.hasNextLine()) {
        String line = scanner.nextLine();
        if (line.trim().length() == 0 || line.startsWith("#")) {
            continue;
        }
        List<String> tokens = tokenize(line);
        Gender gender = Gender.valueOf(tokens.get(0));
        if (gender != null) {
            // type 1 line
            Person person = new Person(tokens.get(1));
            person.setGender(gender);
            addMember(person);
        } else {
            // type 2 line
            Person person = findMember(tokens.get(0));
            for (int i = 1; i < tokens.size(); i++) {
                Person child = findMember(tokens.get(i));
                person.addChild(child);
            }
        }
    }
    scanner.close();
}
```

d) Tegn *objektdiagram* for objektstrukturen som er resultatet av å lese inn eksempelteksten i vedlegg 2 med **load**-metoden til et nyopprettet **Family**-objekt. Hvilke av objektene fantes fra før og hvilke er blitt opprettet av **load**-metoden?

Objektdiagram kan tegnes på mange måter. Det viktigste er å ha med navn med klassen til objektene, attributter med enkle verdier (inkl. String) inni boksen og piler med attributtnavn for referansetyper. Identitet er ikke så viktig her. **Family**- og **Gender**-objektene fantes fra før.



Del 3 – Familierelasjoner og Relation-implementasjoner (25%)

Denne deloppgaven handler om familierelasjoner, som forelder/far/mor, søsken/søster/bror, onkel/tante, søskenbarn/kusine/fetter, niese/nevø osv.

Sister-klassen skal implementere **Relation**-grensesnittet (vedlegg 1), slik at metoden **getRelativesOf** returnerer alle søstrene til **Person**-argumentet. **new Sister().getRelativesOf(person)** skal altså returnere alle søstrene til **person**. Tilsvarende skal klassen **Parent** implementere **Relation** slik at metoden **getRelativesOf** returnerer alle foreldrene, altså mor og/eller far, til **Person**-argumentet. Merk at **Sister**- og **Parent**-klassene *ikke* står i noe vedlegg.

a) Implementer **Sister**- og **Parent**-klassene.

Parent-klassen er enklest. En må få med både mor og far, og unngå å legge til **null**-verdier. I **Sister**-klassen er det laget en hjelpemetode kalt **addChildren**, som legger alle jente-barn av en gitt **Person** til en liste og unngår duplikater. Denne kalles med både mor og far til personen en ønsker å finne søstrene til. Merk at dette også vil legge til halvsøstre. Til slutt fjernes personen selv, for hun er jo ikke sin egen søster (subtilt poeng, jeg vet)!

```
public class Sister implements Relation {

    private void addChildren(Person person, Collection<Person> result) {
          if (person != null) {
                for (Person child : person) {
                      if (child.getGender() == Gender.FEMALE && (!
result.contains(child))) {
```

```
                        result.add(child);
                    }
                }
            }
    }

    @Override
    public Collection<Person> getRelativesOf(Person person) {
        Collection<Person> result = new ArrayList<>();
        addChildren(person.getFather(), result);
        addChildren(person.getMother(), result);
        result.remove(person);
        return result;
    }
}

public class Parent implements Relation {

    @Override
    public Collection<Person> getRelativesOf(Person person) {
        Collection<Person> result = new ArrayList<>();
        if (person.getMother() != null) {
            result.add(person.getMother());
        }
        if (person.getFather() != null) {
            result.add(person.getFather());
        }
        return result;
    }
}
```

Tante-relasjonen kan defineres som søster av forelder, altså en kombinasjonen av **Sister**- og **Parent**-logikken. For å slippe å måtte lage nye klasser for slike sammensatte relasjoner, så defineres **Relation2**-klassen. Den inneholder to **Relation**-objekter **rel1** og **rel2** og implementerer **Relation** slik at metoden **getRelativesOf** returnerer alle som er **rel2**-relatert til de som er **rel1**-relatert til **Person**-argumentet. Hvis **Relation2**-klassen instansieres med **new Relation2(new Parent(), new Sister())**, så vil den i praksis implementere tante-relasjonen ved først å finne foreldrene vha. **Parent**-objektet og så finne foreldrenes søstre vha. **Sister**-objektet.

b) Implementer **Relation2**-klassen. Hva slags standardteknikk er det som brukes her og hva karakteriserer den?

Poenget her er å gjøre som beskrevet i oppgaven, å bruke **rel2** på resultatet av å bruke **rel1**. Her sjekkes det ikke for duplikater (som det sjelden er behov for med relasjoner av denne typen). Dette er *delegeringsteknikken* i praksis, som kjennetegnes ved at et *delegerende* objekt, som skal utføre en oppgave, ber en eller flere *delegater* om å utføre (omtrent) samme oppgave, for så å kombinere resultatene. Her delegeres det til **rel1** og **rel2**, som implementerer samme grensesnitt som **Relation2**. Det er også en form for *komposisjon*, men det er ikke pensum.

```
public class Relation2 implements Relation {
```

```
    public Relation2(Relation rel1, Relation rel2) {
          this.rel1 = rel1;
          this.rel2 = rel2;
    }

    private final Relation rel1, rel2;

    @Override
    public Collection<Person> getRelativesOf(Person person) {
          Collection<Person> result1 = rel1.getRelativesOf(person);
          Collection<Person> result2 = new ArrayList<>();
          for (Person person1 : result1) {
                result2.addAll(rel2.getRelativesOf(person1));
          }
          return result2;
    }
}
```

c) Forklar hvordan **Relation2**-klassen kan brukes til å rigge opp et objekt som implementerer oldeforelder-relasjonen, altså tre foreldre-nivåer unna.

Oldeforelder-relasjonen kan defineres som forelder av forelder av forelder. Dette kan realiseres vha. to (nivåer av) Relation2-objekter (altså uten å skrive en ny klasse!):

```
Relation parent = new Parent();
Relation grandParent = new Relation2(parent, parent);
Relation grandGrandParent = new Relation2(grandParent, parent);
```

Del 4 – Arv og funksjonelle grensesnitt (15%)

a) Hvis en lager **Relation**-implementasjoner for søsken, søster og bror, så vil en finne at de er nesten like. En søster er jo en kvinnelig søsken, mens en bror er en mannlig søsken. Forklar med tekst og kode hvordan en kan lage en **Sibling**-klasse som implementerer søsken-relasjonen, med *subklassene* **Sister** og **Brother** for hhv. søster- og bror-relasjonen.

En kan lage **Sibling**-klassen med utgangspunkt i **Sister**-klassen, og legge til et **gender**-felt som brukes for å plukke ut og legge til barn med den gitte kjønnet (eller begge, hvis **gender** er **null**).Her gjøres det i **addChildren**-metoden, som må endres. **Sister**- og **Brother**-klassene sørger for å initialisere **gender**-feltet med riktig verdi, evt. vha. en **Sister**-konstruktør.
En annen variant er å la **Sister** og **Brother** filtrere superklassen sitt resultat på kjønn, men dette er ikke like god utnyttelse av arv (kunne i prinsippet brukt delegering).

b) Hva er en *abstrakt* klasse? Burde noen av **Sibling**-, **Sister**- eller **Brother**-klassene være abstrakt? Begrunn svaret!

En abstrakt klasse er en klasse som ikke kan instansieres, enten fordi den er ufullstendig ved at den deklarerer én eller flere abstrakte (tomme) metoder, eller fordi det ikke gir mening. Ingen av de tre klassene bør være abstrakte, siden alle er fullstendige og implementerer en nyttig relasjon.

c) Funksjonelle grensesnitt er grensesnitt med noen spesifikke egenskaper, og **Relation** er et slikt grensesnitt. Forklar hvorfor!

Funksjonelle grensesnitt har bare én metode (krav 1), og den metoden er *funksjonell* fordi den for samme input(-parametre) alltid gir samme output(-verdi). En annen måte å si det siste på er at den ikke har intern tilstand som påvirker oppførselen og som kan endres. Det er også vanlig å tenke på grensesnitt-metoden som klassens *hovedfunksjon*.

d) Fullfør deklarasjon under slik at **daughter**-variablen implementerer datter-relasjonen. **daughter.getRelativesOf(person)** skal altså returnere døtrene til **person**.

```
Relation daughter = ... bruk Java 8-syntaks her ...
```

Her brukes uttrykksvarianten av funksjonssyntaksen, siden resultatet beregnes enkelt ved å bruke **getChildren**-metoden. Først har en argumentlista, så en "pil" og så uttrykket som beregner resultatet. Dette omformes til en implementasjon av **Relation** med uttrykket som innholdet i **getRelationOf**-metoden.

```
Relation daughter = (person) -> person.getChildren(Gender.FEMALE);
```

Appendix 1: Provided code (fragments)

```
/**
 * This class represents the gender of a Person.
 * It cannot be instantiated outside this class.
 * It provides all legal Gender values as static variables.
 */
public class Gender {

	... String label;

	... Gender(String label) {
		this.label = label;
	}

	@Override
	public String toString() {
		return label;
	}

	public static Gender
		MALE = new Gender("male"),
		FEMALE = new Gender("female");

	/**
	 * Returns a pre-existing Gender instance for the provided label, or
	 * null of there is no such instance.
	 * @param label
	 * @return a pre-existing Gender instance
	 */
	... Gender valueOf(String label) {
		...
	}
}
```

```java
public class Person implements Iterable<Person> {

	... fields for name, gender, mother and father ...

	... constructor ...

	... methods for name, gender, mother and father ...

	... field(s) for children ...

	/**
	 * @return the number of children of this Person
	 */
	public int getChildCount() {
		...
	}

	/**
	 * @param child
	 * @return if this Person has the provided Person as a child
	 */
	public boolean hasChild(Person child) {
		...
	}

	/**
	 * Returns all children of this Person with the provided Gender.
	 * If gender is null, all children are returned.
	 * Can be used to get all daughters or sons of a person.
	 * @param gender
	 */
	public Collection<Person> getChildren(Gender gender) {
		...
	}

	/**
	 * Adds the provided Person as a child of this Person.
	 * Also sets the child's father or mother to this Person,
	 * depending on this Person's gender.
	 * To ensure consistency, if the provided Person already
	 * has a parent of that gender,
	 * it is removed as a child of that parent.
	 * @param child
	 */
	public void addChild(Person child) {
		...
	}
}
```

```
public class PersonTest extends TestCase {

	public void testAddChild() {
		Gender female = Gender.valueOf("female"),
			male = Gender.valueOf("male");
		Person mother = new Person("Chris"); mother.setGender(female);
		Person father1 = new Person("Pat");  father1.setGender(male);
		Person father2 = new Person("Alex"); father2.setGender(male);
		Person child = new Person("Jean");

		mother.addChild(child);
		assertEquals(1, mother.getChildCount());
		assertTrue(mother.hasChild(child));
		assertEquals(mother, child.getMother());
		mother.addChild(child);
		assertEquals(1, mother.getChildCount());

		father1.addChild(child);
		assertTrue(father1.hasChild(child));
		assertEquals(father1, child.getFather());

		father2.addChild(child);
		assertFalse(father1.hasChild(child));
		assertTrue(father2.hasChild(child));
		assertEquals(father2, child.getFather());

		father2.setGender(female);
		father2.addChild(child);
		// ???
		child.addChild(father2);
		// ???
	}
}
```

```
public class Family {

	/**
	 * Adds a Person as a new family member
	 * @param person the Person to add
	 */
	public void addMember(Person person) {
		...
	}

	/**
	 * Finds a member with the given name
	 * @param name
	 * @return the Person in this Family with the provided name
	 */
	public Person findMember(String name) {
		...
	}

	//

	/**
	 * Writes the contents of this Family to the OutputStream,
	 * so it can be reconstructed using load.
	 * @param out
	 */
	public void save(OutputStream out) ... {
		...
	}

	/**
	 * Helper method that splits a line into a list of tokens,
	 * either words or quoted names (quotes are removed).
	 * @param line – the string to tokenize
	 */
	... List<String> tokenize(String line) {
		... no need to implement this method ...
	}
	/**
	 * Loads contents from the provided InputStream into this Family.
	 * @param in
	 */
	public void load(InputStream in) ... {
		...
	}
}
```

```java
public interface Relation {
	/*
	 * Returns the Collection of Persons related to the provided Person
	 * according to this Relation.
	 * E.g. if this Relation corresponds to the concept of niece,
	 * it should return all Persons that are nieces of person.
	 */
	Collection<Person> getRelativesOf(Person person);
}

public class Relation2 implements Relation {

	public Relation2(Relation rel1, Relation rel2) {
		...
	}

	@Override
	public Collection<Person> getRelativesOf(Person person) {
		...
	}
}
```

Appendix 2: Text format for Family contents

The format is line-based, with three kinds of lines:

1. A line with a gender label followed by a name (in quotes) indicates a new person with that gender and name.
2. Otherwise the line is a sequence of names (in quotes) of persons, the first being the parent of the others (i.e. children).
3. Empty, only whitespace or starting with #. Such lines are ignored when loading.

Lines of all types can be mixed, but there should be no forward references from lines of type 2 to lines of type 1. I.e. the names in lines of type 2 have previously occurred in lines of type 1. Below are two examples of the format.

Example 1:

```
# all persons
# should result in a sequence of calls to Family's addMember method
male "Hallvard Trætteberg"
female "Marit Reitan"
male "Jens Reitan Trætteberg"
female "Anne Trætteberg Reitan"

# all mother/father-child relations
# should result in a sequence of calls to Person's addChild method
"Hallvard Trætteberg" "Jens Reitan Trætteberg" "Anne Trætteberg Reitan"
"Marit Reitan" "Jens Reitan Trætteberg" "Anne Trætteberg Reitan"
```

Example 2:

```
male        "Hallvard Trætteberg"
male "Jens Reitan Trætteberg"
female "Anne Trætteberg Reitan"
"Hallvard Trætteberg" "Jens Reitan Trætteberg" "Anne Trætteberg Reitan"
female      "Marit Reitan"
"Marit Reitan" "Jens Reitan Trætteberg" "Anne Trætteberg Reitan"
```