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.

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

c) Fullfør **valueOf**-metoden.

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.

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

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.

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).

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)

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.

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!

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

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 ble opprettet av **load**-metoden?

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.

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?

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

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.

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

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

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 ...
```

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: 918 97 263

Examination date: 11. June

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: 8 (5+1+2)

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.

An overview of classes and methods for all the parts are provided in appendix 1. The comments contain requirements for the various programming tasks, that must be considered when you solve them. Feel free to define extra methods, in addition to those provided, to make your solution tidier. Useful standard classes and methods can be found in appendix 3.

Part 1 – The Gender and Person classes (30%)

The **Gender** class (appendix 1) represents the gender of a person, which is limited to man and woman for the purpose of this exam. The class must be programmed to make it impossible to have other **Gender** objects than these two. A **Gender** instance has an explanatory label, which is set when the instance is created, and that should not be accessible for other classes.

a) Which *modifiers* are needed in the declaration of the **label** field and constructor? Explain why.

b) How is the class programmed so the explanatory label is shown when output, e.g., using **System.out.println(...)**?

c) Complete the **valueOf** method.

The **Person** class (appendix 1) represents a person, with a name (**String**) and a gender (**Gender**). The name can only be set during creation, while the gender can be set any time.

To handle family relations, the **Person** object includes data about mother, father and children. The **addChild** method is used to relate a child to a parent, and no other methods are specified to modify the child-parent relation. **addChild** both registers the child and sets the child's link to the mother or father, depending on the parent's gender. Given the call **chris.addChild(pat)**, whereby **pat** is registered as the child of **chris**. If **chris** is a *man*, he'll be registered as **pat**'s *father*, whereas if **chris** is a *woman*, she is registered as **pat**'s *mother*.

d) Write fields, methods and constructor for encapsulating **name**, **gender**, **mother** and **father**.

e) Implement **getChildCount**, **hasChild** and **getChildren**, with necessary field(s).

f) As part of the encapsulation of the children relation, we have chosen to let **Person** implement the **Iterable<Person>** interface. How does this affect the code you may write to iterate through the children of a **Person**? Implement the method(s) required by this interface.

g) Implement the **addChild** method, based on the requirements above and in appendix 1, and the requirements that implicitly are tested by **testAddChild** in the **PersonTest** class (appendix 1).

h) At the bottom of the test method **testAddChild** two lines are marked with **// ???** What cases or problems can be tested at these points in the code? How does your code behave, and what do you mean is the correct behaviour in these two cases? (You don't need to correct your code)

Part 2 – The Family class and IO (25%)

The **Family** class (appendix 1) manages persons in a family, with methods for adding family members (**Person** objects), looking up persons by name, saving and loading family members.

a) Write the methods **addMember** and **findMember** (and declare necessary fields), that adds a **Person** as a family member and looks up a member by name, respectively.

In appendix 2 we have described a text format for data about the persons in a family, including the parent-child relation.

b) In appendix 1 there is a helper method name **tokenize**, that can be useful for loading and that can be assumed implemented. What modifier(s) should it have? Explain why!

c) Write the **save** and **load** methods supporting the text format in appendix 2. Explain how (and why) you choose to handle the problem of exceptions.

d) Draw an *object diagram* for the object structure resulting from reading the example text (appendix 2) with the **load** method of a newly created **Family** object. Which of the objects already existed and which ones were created by the **load** method?

Part 3 – Family relations and Relation implementations (25%)

This part concerns family relations, like parent/father/mother, sibling/sister/brother, uncle/aunt, cousins, niece/nephew and so forth.

The **Sister** class should implement the **Relation** interface (appendix 1), so the **getRelativesOf** method returns all the sisters of the **Person** argument. Hence, **new Sister().getRelativesOf(person)** should return all the sisters of **person**. Similarly, the **Parent** class should implement **Relation** so the **getRelativesOf** method returns all the parents, i.e. mother and/or father of the **Person** argument. Note that the **Sister** and **Parent** classes are *not* included in any appendix.

a) Implement the **Sister** and **Parent** classes.

The *aunt* relation can be defined as sister of parent, i.e. a combination of the **Sister** and **Parent** logic. To avoid having to make new classes for such composite relations, we define the **Relation2** class. It contains two **Relation** objects **rel1** and **rel2** and implements **Relation** so the **getRelativesOf** method returns all that are **rel2** related to those that are **rel1** related to the **Person** argument. If the **Relation2** class is instantiated with **new Relation2(new Parent(), new Sister())**, it will effectively implement the aunt relation by first finding the parents using the **Parent** object and then finding the parents' sisters using the **Sister** object.

b) Implementer the **Relation2** class. What standard technique (pattern) is utilized here and what characterizes this technique?

c) Explain how the **Relation2** class can be used to create an object that implements the grand-grand-parent relation, i.e. three parent steps away.

Part 4 – Inheritance and functional interfaces (15%)

a) If you write **Relation** implementations for sibling, sister and brother, you will discover that they are almost the same. A sister is a female sibling, while a brother is a male sibling. Explain with text and code how you can write a **Sibling** class that implements the sibling relation with *sub-classes* **Sister** and **Brother** implementing the sister and brother relations, respectively.

b) What is an *abstract* class? Should any of the **Sibling**, **Sister** or **Brother** classes be abstract? Explain why!

c) Functional interfaces are interfaces with some specific properties, and **Relation** is such an interface. Explain why!

d) Complete the declaration of the **daughter** variable below that implements the daughter relation. I.e. **daughter.getRelativesOf(person)** should return the daughters of **person**.

```
Relation daughter = ... use Java 8 syntax here ...
```

NTNU
Kunnskap for ei betre verd

Institutt for datateknikk og informasjonsvitskap

Eksamensoppgåve i TDT4100
Objektorientert programmering

Fagleg kontakt under eksamen: Hallvard Trætteberg

Tlf.: 918 97 263

Eksamensdato: 11. juni

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 faglærar Hallvard Trætteberg og kvalitetssikra av Ragnhild Kobro Runde (Ifi, UiO).

Målform/språk: Nynorsk

Sidetal: 4

Sidetal vedlegg: 8 (5+1+2)

Kontrollert av:

_____

Dato Sign

Om du meiner at opplysningar manglar i ei oppgåveformulering, gjer kort greie for dei føresetnadene som du finn naudsynte. Om du i ein del er beden om å implementere klasser og metodar og du ikkje klarer det (heilt eller delvis), så kan du likevel nytte dei i seinare delar.

Eit oversyn over klasser og metodar for alle oppgåver er gjeve i vedlegg 1. Kommentarane inneheld krav til dei ulike delane, som du må ta omsyn til når du løyser oppgåva. I tillegg til metodane som er gjevne, står du fritt til å definere ekstra metodar for å gjere løysinga ryddigare. Nyttige standardklasser og -metodar finst i vedlegg 3.

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

Gender-klassen (vedlegg 1) representerer kjønnet til ein person, i denne oppgåva avgrensa til mann og kvinne. Klassen skal kodast slik at det ikkje skal være mogleg å ha andre **Gender**-objekt enn desse to kjønna. Ein **Gender**-instans har eit (forklarande) ord/namn knytt til seg, som settast når instansen lagast, og som ikkje skal vere direkte tilgjengeleg frå andre klasser.

a) Kva for *modifikatorar* bør stå framfor deklarasjonen av **label**-feltet og konstruktøren? Grunngje svaret.

b) Korleis er klassen koda så det forklarande ordet/namnet visast ved utskrift, t.d. med **System.out.println(...)**?

c) Fullfør **valueOf**-metoden.

Person-klassen (vedlegg 1) representerer ein person, med eit namn (**String**) og eit kjønn (**Gender**). Namnet settast berre ved oppretting, medan kjønnet kan settast når som helst.

For å handtere familieforhold så har eit **Person**-objekt data om mor, far og barn. **addChild**-metoden nyttast for å knytte eit barn til en forelder, og det er ikkje spesifisert andre metodar som endrar barn-forelder-koplinga. **addChild** både registrerer barnet og setter barnets kopling til mor eller far, avhengig av kjønnet til forelderen. Tek t.d. kallet **chris.addChild(pat)**, der **pat** vert registert som barnet til **chris**. Om **chris** er *mann*, så blir han registrert som **pat** sin *far*, om **chris** er *kvinne*, så blir ho registrert som **pat** si *mor*.

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

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

f) Som ein del av innkapslinga av barn-koplinga, så har vi vald å la **Person** implementere **Iterable<Person>**-grensesnittet. Kva inneber det for koden ein kan skrive for å gå gjennom barna til ein **Person**? Implementer metoden(e) som er kravd av dette grensesnittet.

g) Implementer **addChild**-metoden, basert på krava som er forklarte over og i vedlegg 1, og krava som implisitt er testa av **testAddChild** i **PersonTest**-klassen (vedlegg 1).

h) Nedst i testmetoden **testAddChild** er to liner markerte med **// ???** Kva for tilfelle eller problem er det som kan testast på dei to punkta i koden? Korleis oppfører din kode seg, og kva meiner du er riktig oppførsel i dei to tilfella? (Du treng ikkje rette på koden din)

Del 2 – Family-klassen og IO (25%)

Family-klassen (vedlegg 1) held orden på person i ein familie, med metodar for å leggje til familiemedlem (**Person**-objekt), slå opp person på namn, lagre familiemedlemmene og lese dei inn igjen.

a) Skriv metodane **addMember** og **findMember** (og definer naudsynte felt), som legg ein **Person** til som familiemedlem og finn eit familiemedlem med et gjeve namn.

I vedlegg 2 blir det gjort greie for eit tekstformat for data om personar i ein familie, inkludert foreldre-barn-koplinga.

b) I vedlegg 1 er det med ein hjelpemetode **tokenize**, som kan vere nyttig når ein les inn og som kan reknast ferdig implementert. Kva for modifikator(ar) burde den ha? Grunngje svaret!

c) Skriv metodane **save** og **load**, som stør dette tekstformatet. Grunngje korleis du vel å behandla problemet med unntak.

d) Teikne *objektdiagram* for objektstrukturen som er resultatet av å lese inn eksempelteksten i vedlegg 2 med **load**-metoden til eit nyoppretta **Family**-objekt. Kva for objekt fanst frå før og kva for objekt vart oppretta av **load**-metoden?

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

Denne deloppgåva handlar om familierelasjonar, som forelder/far/mor, sysken/syster/bror, onkel/tante, syskenbarn/kusine/fetter, niese/nevø osv.

Sister-klassen skal implementere **Relation**-grensesnittet (vedlegg 1), slik at metoden **getRelativesOf** returnerer alle systrene til **Person**-argumentet. **new Sister().getRelativesOf(person)** skal altså returnere alle systrene til **person**. På same vis skal klassen **Parent** implementere **Relation** slik at metoden **getRelativesOf** returnerer alle foreldra, altså mor og/eller far, til **Person**-argumentet. Merk at **Sister**- og **Parent**-klassane *ikkje* står i noko vedlegg.

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

Tante-relasjonen kan definerast som syster av forelder, altså ein kombinasjonen av **Sister**- og **Parent**-logikken. For å sleppe å måtte lage nye klasser for slike samansette relasjonar, så definerast **Relation2**-klassen. Den inneheld to **Relation**-objekt **rel1** og **rel2** og implementerer **Relation** slik at metoden **getRelativesOf** returnerer alle som er **rel2**-relatert til dei som er **rel1**-relatert til **Person**-argumentet. Om **Relation2**-klassen blir instansiert med **new Relation2(new Parent(), new Sister())**, så vil den i praksis implementere tante-relasjonen ved fyrst å finne foreldra vha. **Parent**-objektet og så finne foreldra sine systrer med **Sister**-objektet.

b) Implementer **Relation2**-klassen. Kva for standardteknikk er det som nyttast her og kva karakteriserer den?

c) Forklar korleis **Relation2**-klassen kan nyttast til å rigge opp eit objekt som implementerer oldeforelder-relasjonen, altså tre foreldre-nivå unna.

Del 4 – Arv og funksjonelle grensesnitt (15%)

a) Om ein lager **Relation**-implementasjonar for sysken, syster og bror, så vil ein finne at dei er nesten like. Ei syster er jo eit kvinneleg sysken, medan ein bror er eit mannleg sysken. Forklar med tekst og kode korleis ein kan lage ein **Sibling**-klasse som implementerer sysken-relasjonen, med *subklassane* **Sister** og **Brother** for syster- og bror-relasjonen.

b) Kva er ein *abstrakt* klasse? Burde nokon av **Sibling**-, **Sister**- eller **Brother**-klassane vere abstrakt? Grunngje svaret!

c) Funksjonelle grensesnitt er grensesnitt med nokre spesifikke eigenskaper, og **Relation** er eit slikt grensesnitt. Forklar kvifor!

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

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

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"
```

Appendix 3: Useful standard classes and methods

interface Iterable<T>

void	**forEach**(**Consumer**<? super **T**> action) Performs the given action for each element of the Iterable until all elemen have been processed or the action throws an exception.
Iterator<**T**>	**iterator**() Returns an iterator over elements of type T.

public interface Collection<E> extends Iterable<E>

boolean	**add**(**E** e) Ensures that this collection contains the specified element.
boolean	**addAll**(**Collection**<? extends **E**> c) Adds all of the elements in the specified collection to this collection.
void	**clear**() Removes all of the elements from this collection.
boolean	**contains**(**Object** o) Returns true if this collection contains the specified element.
boolean	**containsAll**(**Collection**<?> c) Returns true if this collection contains all of the elements in the specified collection.
boolean	**isEmpty**() Returns true if this collection contains no elements.
boolean	**remove**(**Object** o) Removes a single instance of the specified element from this collection, if it is present.
boolean	**removeAll**(**Collection**<?> c) Removes all of this collection's elements that are also contained in the specified

	collection.
boolean	**removeIf**(**Predicate**<? super **E**> filter) Removes all of the elements of this collection that satisfy the given predicate
boolean	**retainAll**(**Collection**<?> c) Retains only the elements in this collection that are contained in the specified collection.
int	**size**() Returns the number of elements in this collection.
Stream<**E**>	**stream**() Returns a sequential Stream with this collection as its source.

interface List<E> extends Collection<E>

void	**add**(int index, **E** element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(int index, **Collection**<? extends **E**> c) Inserts all of the elements in the specified collection into this list at the specified position.
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(**Object** o) Returns the index of the first occurrence of the specified element in this list, or -1 if it does not contain the element.
int	**lastIndexOf**(**Object** o) Returns the index of the last occurrence of the specified element in this list, or -1 if it does not contain the element.
E	**remove**(int index) Removes the element at the specified position in this list.
E	**set**(int index, **E** element) Replaces the element at the specified position in this list with the specified element.
Void	**sort**(**Comparator**<? super **E**> c) Sorts this list according to the order induced by the specified **Comparator**.

interface Map<K,V>

void	**clear**() Removes all of the mappings from this map.
boolean	**containsKey**(**Object** key) Returns true if this map contains a mapping for the specified key.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for t key.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Set<**K**>	**keySet**() Returns a **Set** view of the keys contained in this map.
V	**put**(**K** key, **V** value) Associates the specified value with the specified key in this map.
void	**putAll**(**Map**<? extends **K**,? extends **V**> m) Copies all of the mappings from the specified map to this map.
V	**remove**(**Object** key) Removes the mapping for a key from this map if it is present.
int	**size**() Returns the number of key-value mappings in this map.

interface Stream<T>

boolean	**allMatch**(**Predicate**<? super **T**> predicate) Returns whether all elements of this stream match the provided predica
boolean	**anyMatch**(**Predicate**<? super **T**> predicate) Returns whether any elements of this stream match the provided predicate.
<R,A> R	**collect**(**Collector**<? super **T**,A,R> collector) Performs a **mutable reduction** operation on the elements of this stream using a Collector.
Stream<**T**>	**filter**(**Predicate**<? super **T**> predicate) Returns a stream consisting of the elements of this stream that match the given predicate.
void	**forEach**(**Consumer**<? super **T**> action) Performs an action for each element of this stream.
Stream<R>	**map**(**Function**<? super **T**,? extends R> mapper) Returns a stream consisting of the results of applying the given function to the elements of this stream.
T	**reduce**(**T** identity, **BinaryOperator**<**T**> accumulator) Performs a **reduction** on the elements of this stream, using the provided identity value and an **associative** accumulation function, and returns the reduced value.
Stream<**T**>	**sorted**() Returns a stream consisting of the elements of this stream, sorted according to natural order.
Stream<**T**>	**sorted**(**Comparator**<? super **T**> comparator) Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.

class String implements Comparable<String>

char	**charAt**(int index) Returns the char value at the specified index.
boolean	**contains**(**String** s) Returns true if and only if this string contains the specified string.
boolean	**endsWith**(**String** suffix) Tests if this string ends with the specified suffix.
static **String**	**format**(**String** format, **Object**... args) Returns a formatted string using the specified format string and arguments.
int	**indexOf**(int ch) Returns the index within this string of the first occurrence of the specified character.

int	**indexOf**(int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified characte starting the search at the specified index.
int	**indexOf**(**String** str) Returns the index within this string of the first occurrence of the specified substring.
int	**indexOf**(**String** str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
boolean	**isEmpty**() Returns true if, and only if, **length()** is 0.
int	**lastIndexOf**(int ch) Returns the index within this string of the last occurrence of the specified character.
int	**lastIndexOf**(int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
int	**lastIndexOf**(**String** str) Returns the index within this string of the last occurrence of the specified substring.
int	**lastIndexOf**(**String** str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int	**length**() Returns the length of this string.
boolean	**regionMatches**(int toffset, **String** other, int ooffset, int len) Tests if two string regions are equal.
String	**replace**(char oldChar, char newChar) Returns a string resulting from replacing all occurrences of oldChar in this string with newChar.
String	**replace**(**String** target, **String** replacement) Replaces each substring of this string that matches the literal target string with the specified literal replacement string.
String[]	**split**(**String** regex) Splits this string around matches of the given **regular expression**.
boolean	**startsWith**(**String** prefix) Tests if this string starts with the specified prefix.
String	**substring**(int beginIndex) Returns a string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a string that is a substring of this string.
String	**toLowerCase**() Converts all of the characters in this String to lower case using the rules of the default locale.
String	**toUpperCase**() Converts all of the characters in this String to upper case using the rules of the default locale.
String	**trim**() Returns a string whose value is this string, with any leading and trailing whitespace removed.

class Scanner

Scanner(**InputStream** source) Constructs a new Scanner that produces values scanned from the specified input stream.	
void	**close**() Closes this scanner.
boolean	**hasNext**() Returns true if this scanner has another token in its input.
boolean	**hasNextBoolean**() Returns true if the next token in this scanner's input can be interpreted as a boolean value using a case insensitive pattern created from the string "true|false".
boolean	**hasNextDouble**() Returns true if the next token in this scanner's input can be interpreted as a double value using the **nextDouble()** method.
boolean	**hasNextInt**() Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the **nextInt()** method.
boolean	**hasNextLine**() Returns true if there is another line in the input of this scanner.
String	**next**() Finds and returns the next complete token from this scanner.
boolean	**nextBoolean**() Scans the next token of the input into a boolean value and returns that value.
double	**nextDouble**() Scans the next token of the input as a double.
int	**nextInt**() Scans the next token of the input as an int.
String	**nextLine**() Advances this scanner past the current line and returns the input that was skipped.