NTNU
Kunnskap for en bedre verden

Institutt for datateknikk og informasjonsvitenskap

Kontinuasjonseksamensoppgave i TDT4100 Objektorientert programmering

Faglig kontakt under eksamen: Hallvard Trætteberg

Tlf.: 91897263

Eksamensdato: 3. august

Eksamenstid (fra-til): 9.00-13.00

Hjelpemiddelkode/Tillatte hjelpemidler: C

Kun ”Big Java”, av Cay S. Horstmann, er tillatt.

Annen informasjon:

Oppgaven er utarbeidet av faglærer Hallvard Trætteberg og kvalitetssikret av Ragnhild Kobro Runde (Ifi, UiO).

Målform/språk: Bokmål

Antall sider:

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.

Strukturen av kode og metoder det skal jobbes med på denne eksamen, er samlet i vedlegget. Oppførselen er spesifisert i strukturerte kommentarer over klasser og metoder. Det blir henvist til vedlegget i hele oppgavesettet.

Del 1 – Unit-klassen (35%)

a) **Unit**-klassen er utformet slik at instanser ikke skal kunne endres etter at de er opprettet. Hva er *generelle* fordeler og ulemper med klasser som gir ikke-modifiserbare (immutable) instanser?

Klassen blir enklere, og minsker bl.a. behovet for validering. Instanser kan brukes av flere deler av et program, uten risiko for at en del endrer på dem og ødelegger for en annen del. Ulempen er at en må lage nye instanser hvis de må rettes på, istedenfor å endre dem direkte.

b) Skriv ferdig de tre konstruktørene og definer nødvendige felt. Hva er hensikten med bruken av **throws**-nøkkelordet, slik det er brukt her? Er det strengt tatt nødvendig og evt. hvorfor/hvorfor ikke?

```
	private final String symbol;

	public Unit(String symbol) throws IllegalArgumentException {
		this(symbol, null, 1.0, 0.0);
	}

	private final Unit base;
	private final double factor, offset;

	public Unit(String symbol, Unit base, double factor, double offset)
throws IllegalArgumentException {
		for (int i = 0; i < symbol.length(); i++) {
			char c = symbol.charAt(i);
			if (! Character.isAlphabetic(c)) {
				throw new IllegalArgumentException(c + " is an illegal
symbol character");
			}
		}
		this.symbol = symbol;
		this.base = base;
		this.factor = factor;
		this.offset = offset;
	}

	public Unit(String symbol, Unit base, double factor) throws
IllegalArgumentException {
		this(symbol, base, factor, 0.0);
	}
```

throws-deklarasjonen forteller leseren av koden at konstruktørene kan utløse unntak. Siden unntakene er en subklasse av **RuntimeException** og dermed ikke *checked*, så er det ikke nødvendige.

c) Hva er hensikten med å definere en **toString()**-metode?

toString()-metoden brukes implisitt når Java lager String-objekter av instanser ifm. bruk av + og IO og sikrer at tilstanden til instanser blir presentert på en nyttig måte.

d) Nederst i klassen defineres en del meter-relaterte Unit-instanser (**m**, **km** og **dm** og **cm)**. Tegn objektdiagram som illustrerer objektstrukturen som disse instansene utgjør.

Her er verdiene til feltene og kjeden av base-linker vesentlig.

e) Metoden **findCommonBaseUnit** er sentral i konvertering av verdier mellom ulike enheter. Den skal virke slik at **dm.findCommonBaseUnit(km)** returnerer **m**-instansen. Skriv ferdig metoden.

```
public Unit findCommonBaseUnit(Unit other) {
      Unit unit1 = this;
      while (unit1 != null) {
            Unit unit2 = other;
            while (unit2 != null) {
                  if (unit2 == unit1) {
                        return unit1;
                  }
                  unit2 = unit2.base;
            }
            unit1 = unit1.base;
      }
      return null;
}
```

f) **convert**-metoden er ferdigskrevet og bruker de to hjelpemetodene **convertToBase** og **convertFromBase**. De to hjelpemetodene kaller også seg selv. Forklar hvilke kall som gjøres til disse (inkludert de til seg selv) og hvilke argumenter de får og verdier de returnerer, i løpet av utførelsen av **dm.convert(2.0, km)**.

…

g) **valueOf**-metoden ”oversetter” fra et enhetssymbol til en **Unit**-instans, litt på samme måten som **Double.valueOf** lager en **Double**-verdi fra en **String**. Men merk at **Unit.valueOf** ikke skal lage nye instanser, men returnere en av de predefinerte! Skriv ferdig **valueOf**-metoden.

```
public static Unit valueOf(String symbol) {
      for (Unit unit : ALL_UNITS) {
            if (symbol.equals(unit.symbol)) {
                  return unit;
            }
      }
      return null;
}
```

Del 2 –Value-klassen (25%)

a) Hva er hensikten med get-metoder? Fullfør konstruktøren, get-metodene og deklarer nødvendige felt.

get-metoder gir tilgang til tilstanden uten å "avsløre" hvilke felt som brukes for å representere tilstanden.

```
	private final Unit unit;
	private final double value;

	public Value(Unit unit, double value) {
		this.unit = unit;
		this.value = value;
	}

	public Unit getUnit() {
		return unit;
	}

	public double getValue() {
		return value;
	}
```

b) **valueOf**-metoden "oversetter" fra en **String** til en **Value**-instans, litt på samme måten som **Double.valueOf** lager en **Double**-verdi fra en **String**. Skriv ferdig **valueOf**-metoden.

```
	public static Value valueOf(String s) {
		int pos = s.length();
		while (Character.isAlphabetic(s.charAt(pos - 1))) {
			pos--;
		}
		return new Value(Unit.valueOf(s.substring(pos)),
Double.valueOf(s.substring(0, pos)));
	}
```

c) Skriv ferdig **add**, **compute** og **mult**-metodene.

```
	public Value add(Value other) {
		Unit base = this.unit.findCommonBaseUnit(other.unit);
		double sum = this.unit.convert(value, base) +
other.unit.convert(other.value, base);
		return new Value(base, sum);
	}

	public Value compute(BinaryOperator<Double> op, double other) {
		double result = op.apply(this.value, other);
		return new Value(this.unit, result);
	}

	public Value mult(double other) {
```

```
            return compute((v1,  v2) -> v1 * v2, other);
      }
```

d) **Value**-klassen implementerer **Comparable<Value>**. Hvorfor er det nyttig? Skriv ferdig **compare**-metodene.

Ved å implementere **Comparable**-grensesnittet så kan **Value**-objekter sorteres vha. Java sine innebygde sort-metoder.

```
      public int compareTo(Value other) {
            Unit base = this.unit.findCommonBaseUnit(other.unit);
            if (base == null) {
                  throw new IllegalArgumentException("Cannot compare " + this +
" with " + other);
            }
            double d1 = this.unit.convert(value, base), d2 =
other.unit.convert(other.value, base);
            if (d1 < d2) {
                  return -1;
            } else if (d1 > d2) {
                  return 1;
            } else {
                  return 0;
            }
      }
```

Del 3 – Values-grensesnittet og ValueSeries-klassen (25%)

a) **Values**-grensesnittet utvider **Iterable<Double>**. Hvorfor er det nyttig/praktisk?

b) **Values**-grensesnittet er ment å støtte *observerbarhet* og observert-observatør-teknikken, men deklarasjonen av metodene for lytterhåndtering er erstattet av "…". Skriv de to deklarasjonene.

```
      public void addValuesListener(ValuesListener listener);
      public void removeValuesListener(ValuesListener listener);
```

c) Skriv ferdig konstruktøren og de to **appendValue**-metodene i **ValueSeries**-klassen. Definer nødvendige felt.

```
      private final Unit unit;
      private Collection<Double> values  = new ArrayList<Double>();

      public ValueSeries(Unit unit) {
            this.unit = unit;
      }

      public void appendValue(double value) {
            values.add(value);
            fireValuesChanged();
      }
```

```
	public void appendValue(Value value) {
		appendValue(value.getUnit().convert(value.getValue(), getUnit()));
	}
```

d) Implementer metodene i **ValueSeries**-klassen, som er nødvendige for å gjøre klassen komplett.

```
	@Override
	public int size() {
		return values.size();
	}

	@Override
	public double average() {
		double sum = 0.0;
		for (double value : values) {
			sum += value;
		}
		return sum / values.size();
	}

	@Override
	public Unit getUnit() {
		return unit;
	}

	@Override
	public Iterator<Double> iterator() {
		return values.iterator();
	}

	//

	@Override
	public Values add(Values other) {
		Unit base = this.unit.findCommonBaseUnit(other.getUnit());
		ValueSeries result = new ValueSeries(base);
		Iterator<Double> otherDoubles = other.iterator();
		for (double value : values) {
			double otherDouble = otherDoubles.next();
			double sum = this.unit.convert(value, base) +
other.getUnit().convert(otherDouble, base);
			result.appendValue(sum);
		}
		return result;
	}

	// ValuesListener support

	private Collection<ValuesListener> listeners = new
ArrayList<ValuesListener>();

	@Override
```

```
	public void addValuesListener(ValuesListener listener) {
		listeners.add(listener);
	}

	@Override
	public void removeValuesListener(ValuesListener listener) {
		listeners.remove(listener);
	}

	protected void fireValuesChanged() {
		for (ValuesListener listener : listeners) {
			listener.valuesChanged(this);
		}
	}
```

Del 4 – Testing (10%)

I denne delen skal du skrive testkode for **Value**- og **ValueSeries**-klassene. Hensikten er å vise at du behersker testmetodikken, og du kan, men trenger ikke bruke JUnit-rammeverket.

a) Skriv testkode for **Value** sin **valueOf**-metode.

```
	public void testValueOf() {
		Value value = Value.valueOf("2.0m");
		assertEquals(Unit.valueOf("m"), value.getUnit());
		assertEquals(2.0, value.getValue());
	}
```

b) Forklar med tekst og kode hvordan du kan teste at **ValueSeries** håndterer lyttere (av typen **ValuesListener**) og implementerer observerbarhet riktig.

```
public class ValueSeriesTest extends TestCase implements ValuesListener {

	private ValueSeries values;

	protected void setUp() throws Exception {
		super.setUp();
		values = new ValueSeries(new Unit("m"));
		values.addValuesListener(this);
	}

	public void testValuesChanged() {
		values.appendValue(11);
		assertEquals(values, notified);
		notified = null;
		values.removeValuesListener(this);
		values.appendValue(12);
		assertEquals(null, notified);
	}

	private Values notified = null;
```

```
	@Override
	public void valuesChanged(Values values) {
		notified = values;
	}
}
```

Appendix: Specification of interfaces and classes and their methods. Some implementation details are provided, while other details are not given (since these are your task).

```
package kont2015;

/**
 * A class for (scientific) units like meter (m), grams (g) and Kelwin (K).
 * Units may be derived from each other by a linear formula, e.g. km from m and C
(Celcius) from Kelwin.
 * @author hal
 *
 */
public class Unit {

	/**
	 * Constructor for base units, e.g. meter, gram, kelwin etc.
	 * Initialises with the symbol, e.g. "m" for meters, "g" for grams, "K" for
kelwin.
	 * @param symbol The symbol of this Unit, must contain only alphabetic
characters.
	 * @throws IllegalArgumentException if the symbol contains characters that
are not alphabetic
	 */
	public Unit(String symbol) throws IllegalArgumentException {
		...
	}

	/**
	 * Constructor for derived units, e.g. kilometer, milligram and Celcius,
derived from meter, gram and Kelwin respectively.
	 * A derived unit includes the factor and offset for the linear formula for
computing the base unit from the derived one.
	 * base-unit-value = derived-unit-value * factor + offset
	 * @param symbol The symbol for the derived Unit
	 * @param base The base unit, e.g. meter for kilometer, gram for milligram
	 * @param factor The factor in the formula, e.g. 1000 for km to m or 0.001
for mg to g.
	 * @param offset The offset in the formula.
	 * @throws IllegalArgumentException if the symbol contains characters that
are not alphabetic
	 */
	public Unit(String symbol, Unit base, double factor, double offset)
throws IllegalArgumentException {
		...
	}

	/**
	 * Constructor for derived units, e.g. kilometer, milligram and Celcius,
derived from meter, gram and Kelwin respectively.
	 * A derived unit includes the factor and offset for the linear formula for
computing the base unit from the derived one.
```

```
     * base-unit-value = derived-unit-value * factor + offset
     * @param symbol The symbol for the derived Unit
     * @param base The base unit, e.g. meter for kilometer, gram for milligram
     * @param factor The factor in the formula, e.g. 1000 for km to m or 0.001
for mg to g. The offset is set to 0.0.
     * @throws IllegalArgumentException if the symbol contains characters that
are not alphabetic
     */
    public Unit(String symbol, Unit base, double factor) throws
IllegalArgumentException {
        ...
    }

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

    /**
     * Finds the first common unit from which both this and the other Unit is
derived.
     * If other is derived from this, then this is returned, or if this is
derived from other, then other is returned.
     * Otherwise it finds the first base unit that both are derived from.
     * @param other The other unit.
     * @return The first common unit that is a common base unit for both this
and other.
     */
    public Unit findCommonBaseUnit(Unit other) {
        ...
    }

    /**
     * Converts value from this unit to the other unit.
     * @param value The value to convert.
     * @param other The other unit, that value is converted to.
     * @return value converted from this unit to the other unit
     * @throws IllegalArgumentException if there is no common base unit.
     */
    public double convert(double value, Unit other) throws
IllegalArgumentException {
        Unit base = findCommonBaseUnit(other);
        if (base == null) {
            throw new IllegalArgumentException("Cannot convert from " +
this + " to " + other);
        }
        double baseValue = convertToBase(value, base);
        return other.convertFromBase(baseValue, base);
    }

    /**
     * Helper method for converting from this unit to a specific base unit.
```

```
     * @param value The value to convert.
     * @param base The base unit to convert to.
     * @return The converted value.
     */
    private double convertToBase(double value, Unit base) {
        if (this == base) {
            return value;
        }
        if (this.base == null) {
            throw new IllegalArgumentException(base + " is not a base for
" + this);
        }
        return this.base.convertToBase(value * factor + offset, base);
    }

    /**
     * Helper method for converting from a specific base unit to this unit.
     * @param value The value to convert.
     * @param base The base unit to convert from.
     * @return The converted value.
     */
    private double convertFromBase(double value, Unit base) {
        if (this == base) {
            return value;
        }
        if (this.base == null) {
            throw new IllegalArgumentException(base + " is not a base for
" + this);
        }
        return this.base.convertFromBase(value, base) / factor - offset /
factor;
    }

    // The currently supported predefined units, that are considered by the
valueOf method
    private static Unit
        m = new Unit("m"),
        km = new Unit("km", m, 1000.0),
        dm = new Unit("dm", m, 0.1),
        cm = new Unit("cm", dm, 0.1),
        ALL_UNITS[] = {m, km, dm, cm};

    /**
     * Finds the Unit for the given symbol among all predefined units.
     * Currently supported units are m, km, dm, cm
     * @param symbol the symbol to search for, e.g. "m" or "dm"
     * @return the Unit with the given symbol, or null, of no such Unit was
found
     */
    public static Unit valueOf(String symbol) {
        ...
    }
```

```
}
```

```
package java.util.function;

/**
 * Represents a function that accepts two arguments and produces a result.
 * This is the two-arity specialization of {@link Function}.
 */

public interface BinaryOperator<T> {

    /**
     * Applies this function to the given arguments.
     *
     * @param t the first function argument
     * @param u the second function argument
     * @return the function result
     */
    T apply(T t1, T t2);
}
```

```
package kont2015;

import java.util.function.BinaryOperator;

/**
 * A class representing a double value in a certain Unit.
 * Instances are immutable, i.e. cannot be modified once created.
 * @author hal
 *
 */
public class Value implements Comparable<Value> {

      /**
       * Creates a new value with the provided Unit and (double) value.
       * @param unit the Unit of the new Value
       * @param value the double value of the new Value
       */
      public Value(Unit unit, double value) {
            ...
      }

      @Override
      public String toString() {
            return getValue() + unit.toString();
      }

      /**
       * Returns this Value's Unit
       * @return this Value's Unit
       */
      ... get... {
```

```
        ...
    }

    /**
     * Returns this Value's double value
     * @return this Value's double value
     */
    ... get... {
        ...
    }

    /**
     * Returns a Value instance from the provided String.
     * The format is a double (as parsed by Double.valueOf) followed by a Unit
symbol (as parsed by Unit.valueOf), e.g. "1.0m" or "2.5km".
     * @param s
     * @return the String parses as a Value instance
     */
    public static Value valueOf(String s) {
        ...
    }

    /**
     * Computes the sum of this and other (both Value objects).
     * The Unit of the returned Value is the common base Unit of this and
other's Units.
     * Hence, both Value object's double values are properly converted before
adding.
     * @param other the other Value
     * @return a new Value object representing the sum of this and other
     */
    public Value add(Value other) {
        ...
    }

    /**
     * Computes a new value that is the combination of this Value's double
value and the provided double.
     * The double values are combined using the provided BinaryOperator.
     * The Unit of the returned Value is the this Value's Unit.
     * @param other the double value to combine with this
     * @return a new Value object representing the sum of this and other
     */
    public Value compute(BinaryOperator<Double> op, double other) {
        ...
    }

    /**
     * Computes the product of this Value and other (a double) using the
compute method.
     * @param other the other factor
     * @return the product of this Value's double value and other (also a
```

```
double)
	 */
	public Value mult(double other) {
		...
	}

	/**
	 * Compares this Value with other according the the requirements of
Comparable.
	 * Note that this Value and other may have different Units.
	 * @throws IllegalArgumentException if this and other don't have a common
base Unit.
	 */
	@Override
	public int compareTo(Value other) {
		...
	}
}
```

```
package kont2015;

/**
 * Interface implemented by classes wanting to be notified of changes to a Values
object.
 * @author hal
 *
 */
public interface ValuesListener {
	public void valuesChanged(Values values);
}
```

```
package kont2015;

/**
 * An interface representing a sequence of doubles, all with the same Unit.
 * Implementing classes may support adding and removing elements. If modifiable,
 * the class must support notifying ValuesListeners.
 * @author hal
 *
 */
public interface Values extends Iterable<Double> {

	/**
	 * @return this Values' Unit
	 */
	public Unit getUnit();

	/**
	 * @return the number of values in the sequence.
	 */
	public int size();
```

```
	/**
	 * Computes the average of this Values object's double values.
	 * @return the average as a double (implicitly in this Values' Unit).
	 */
	public double average();

	/**
	 * Creates a new Values object where each element is the sum of
corresponding elements in this and the other Values object.
	 * @param other the other Values
	 * @return a new Values object that represents the sum of this and other.
	 */
	public Values add(Values other);

	// ValuesListener support

	/**
	 * Adds a ValuesListener to be notified of changes to this Values object.
	 * @param listener
	 */
	...

	/**
	 * Removes a previously added ValuesListener.
	 * @param listener
	 */
	...
}

package kont2015;

/**
 * A class representing series of doubles, all with the same (unmodifiable) Unit.
 * A Values object can only be modified by appending another value.
 * @author hal
 *
 */
public class ValueSeries implements Values {

	/**
	 * Constructs a new ValueSeries object, with the provided Unit.
	 * @param unit
	 */
	public ValueSeries(...) {
		...
	}

	/**
	 * Appends the provided double to this ValueSeries. The double value is
assumed to be in the Unit of this ValueSeries.
	 * @param value the double to append
```

```
	 */
	public void appendValue(double value) {
		...
	}

	/**
	 * Appends the provided Value to this ValueSeries, by first converting it
to this ValueSeries' unit and then appending it.
	 * @param value the Value to append, after converting it to this
ValueSeries' Unit.
	 */
	public void appendValue(Value value) {
		...
	}

	// Values methods

	...

}
```