com.schneide.quantity V0.9 HOWTO

Matthias Kempka (Matthias.Kempka@softwareschneiderei.de)

December 22, 2003

Contents

1 Quick start

For the impatient, here is how to quickly use the Java Quantity Framework:

- Make sure to have the Java Quantity files in your classpath, for example, include the jar file in your classpath.
- Make use of the library as it is shown in the following code fragment, which adds 3600 seconds to an existing hour object, and computes the average speed in meters per second over a length of 100 Kilometer.

```
Hour time = new Hour(1);
time.add(new Second(3600));
Kilometer length = new Kilometer(100);
Quantity averageSpeed = Quantity.divide(length, time);
MeterPerSecond speed = SpeedHandler.changeToMeterPerSecond(
                                       averageSpeed);
System.out.println("Average speed: " + speed.toString());
```

If you need different quantities than stated in this example, just try to type them in. The library contains about 100 physical units in different scales.

2 Definitions of words used in this document

Atomar Signature An *atomar signature* describes a physical dimension.
Examples: Length, Weight

Signature A *signature* is any multiplication or fraction of atomar signatures. Every atomar signature is a signature.
Examples: Speed (= $\frac{Length}{Time}$), Acceleration (= $\frac{Length^2}{Time}$)

Atomar Unit An *atomar unit* is a measure associated with an atomar signature.
Examples: Meter describes Length, Second describes Time

Scaled Unit A *scaled unit* is a scaled atomar signature. Every atomar unit is a scaled unit.
Examples: Millimeter, Kilogram

Compound Unit A *compound unit* is a combination of scaled units. Compound units are a measurement for the associated signature. Every scaled unit and every atomar unit is a compound unit.
Examples: $\frac{Meter}{Second}$ and $\frac{Kilometer}{Hour}$ describe Speed, Newton = $\frac{Meter^2 \cdot Kilogram}{Second}$ describes Force.

Quantity An *quantity* combines a value with a compound unit.
Examples: $2\frac{Meter}{Second}$, $4 \cdot 10^{12} Nanoseconds$

3 Non-ambiguous quantities

The most common use case of quantities is the use of a non-ambiguous quantity, i.e. the well defined statement of the scale unit and metric of the quantity. This is done by using one of the many provided classes found in the packages

```
com.schneide.quantity.electricalQuantities
com.schneide.quantity.mechanicalQuantities
com.schneide.quantity.radioactivityQuantities
com.schneide.quantity.miscQuantities
```

Currently, only physical units are provided. The package and class names should speak for themselves.

The unit of a non-ambiguous quantity can not be changed.

Example If you want to use a quantity with the unit kilogram, you simply write

```
Kilogram twoKilogram = new Kilogram(2);
```

to get an instance of Kilogram in your program.

Check the Javadoc documentation to see what quantities are available.

4 Ambiguous quantities

The package `com.schneide.quantity` contains a class named `Quantity`. This is a quantity that does not contain a specific unit, but can describe units of different kinds. Objects of its type can contain any units including the dimensionless unit. In the constructor you have to specify the value and the unit of the quantity. Objects of the class Quantity can change the unit during their lifetime.

Commonly used units can be achieved from the class `FrequentlyUsedUnits`.

Example You can create a Quantity representing 2 Kilogram with the following expression:

```
Quantity twoKgQuantity = new Quantity(2,
        FrequentlyUsedUnits.getKilogram);
```

Multiplication of quantities is implemented in the class `Quantity`. It always returns an object of the type `Quantity`.

4.1 Use cases for ambiguous quantities

Originally, the library was created to make the interfaces in a large-scaled program clear. It reduces the number of `double` parameters and confusion of the scale that usually comes with those interfaces. We strongly recommend to make use of the library in this way, if you pass quantities from GUI components toward other components in your project, especially if you work with the quantities.

There is, however, a different approach that says, “I, the programmer, don’t care”, meaning the user types in some values of some type, you save it in an ambiguous `Quantity` object and at the time of giving the quantity back to the user you print out the value and the unit string and that's it. It is convenient to use this approach if the quantity in question just has to be saved without working much with it. Be careful which approach you use. You may run into great debugging difficulties if you have larger calculations in your program and just have `Quantity` objects.

5 Conversion between similar quantities

Quantities with the same signature can be converted into each other, this means, since $\frac{Meter}{Second}$ and $\frac{Kilometer}{Second}$ both describe Speed, the latter can be converted into the former and vice versa.

There are even conversion methods for not-standard-conforming units like `DegreeFahrenheit` and `DegreeCelsius`, although those units can not be calculated with.

The conversion methods are static methods in specialized handlers. There is a handler for each signature. The `LengthHandler`, for example is able to convert lengths, that are `Meter`, `Millimeter`, `Kilometer` and so on. The `CurrentHandler` is able to convert `Ampere` and `Microampere` while the `TimeHandler` converts `Hour` and `Seconds` and so on.

The handlers can be found in additional packages `*.handler` below the non-ambiguous quantity class packages.

Example Calling the method `TimeHandler.convertToHour(AbstractTime)` will return a new `Hour` object with the converted value of the original `AbstractTime` object. This can be any inheritant of `AbstractTime` like `Second`, `Minute` or `Hour`.

```
Second manySeconds = new Second(3600);
Hour oneHour = TimeHandler.convertToHour(manySeconds);
```

The object `oneHour` will now contain the value 1.

Be aware that conversions may cause rounding errors.

6 Conversion between ambiguous and non-ambiguous quantities

Similar to the conversion described in 5 the QuantityHandler turns any non-ambiguous quantities to a Quantity.

If you have an ambiguous Quantity, and you know at a specific time what kind of unit dimension it describes, it can be converted to a non-ambiguous quantity. The methods will throw a `WrongUnitException` if the signature of the quantity in question does not match the signature of the used handler.

Example After a basic arithmetic operation you usually know what kind of unit your quantity describes.

```
MeterPerSecond mySpeed = new MeterPerSecond(2);
Second myTime = new Second(1);
Quantity multResult = Quantity.multiply(mySpeed, myTime);
// now you know that multResult is describing a length
Meter wayPassedInTime = LengthHandler.convertToMeter(multResult);

// the following will throw a WrongUnitException:
Kilogram falseUnit = WeightHandler.convertToKilogram(multResult);
```

7 Arithmetic operations with quantities:

As stated earlier, the quantities support the four basic arithmetic operations. Addition and substraction can only be done with quantities of the same signature while multiplication and division usually results in a different signature. This is why they are treated differently.

7.1 Addition and Substraction

Addition and substraction are mostly the same for ambiguous and non-ambiguous quantities with the single difference that an operation with ambiguous quantities may throw a `WrongUnitException` while you will get compile-time errors if you try to operate with non-ambiguous quantities with different signatures.

There are two ways to do an addition or substraction.

- You can use the object methods `add()` and `substract()` which will add respectively substract the argument to or from the object. The specific instanciation of the dimension is not important.

  ```
  Second manySeconds = new Second(3600);
  Hour myHour = new Hour(1);
  myHour.add(manySeconds);
  ```

- Every non-ambiguous quantity class has static methods for addition and substraction. They return a newly created object of this class.

  ```
  Second threeHoursInSeconds = Second.add(myHour,  manySeconds);
  ```

  There exist special quantities that describe values of a specific signature but that are not able to be used in arithmetic operations. This is due to undefined behaviour in arithmetic operations. For example, in the vernacular it seems clear that

$$2^oC + 2^oC = 4^oC$$

But if you go to the definition it is

$$2^oC + 2^oC = 275K + 275K = 550K = 277^oC$$

Because of this unclearness in expected behaviour we decided not to allow such an operation with the quantities in question. If a calculation with such a quantity is wished, they can be converted to their SI-units which can be calculated with.

7.2 Multiplication and Division

Every multiplication with quantities different from the dimensionless quantity (which is synonymous with a scalar) will leave the signature of the two multiplicated quantities. This is why for multiplications the ambiguous quantity class 'Quantity' has to be used.

We recommend to convert a resulting Quantity as soon as possible after multiplications and divisions to not loose the overview of the contents of the object. See Section 6 to learn how this is done.

Example

```
Newton myForce = new Newton(2);
Second myTime = new Second(2);
KilometerPerHour mySpeed = new KilometerPerHour(7.2);
// multiplication with quantities:
Quantity temporaryQuantity = Quantity.multiply(mySpeed, myTime);
temporaryQuantity.multiply(myForce);
Joule result = EnergyHandler.convertToJoule(temporaryQuantity);
// multiplication with a scalar:
result.multiply(2);
```

8 Serializing and recovering quantities

Every quantity has the possibility to create a recreationable String and an appropriate constructor to create a new quantity from that string. This constructor throws a `WrongUnitException` if the string describes a different unit than the quantity that created the string even if it is the same signature.

The equivalent constructor of the ambiguous quantity works with every recreational String.

Example

```
Ampere myAmpere = new Ampere(7.2);
String myRecreationableString = myAmpere.toRecreationableString();
Ampere recoveredAmpere = new Ampere(myRecreationableString);
```

9 Creating new quantities

For physical SI quantities the package should be quite complete. However, you might need quantities that we did not think of.

The superclass of all quantity classes is the class `AbstractQuantity`. It contains all the functionality to calculate with and convert quantities. It is tightly coupled with the class `CompoundUnit`. Together the class `AbstractQuantity` and `CompoundUnit` contain the functionality of the framework. Inherited classes bring the functionality to the user in form of specialized quantities.

A *Quantity* combines a value with a unit. More specific, a Quantity of the library `com.schneide.quantity` contains 3 members called `value`, `power` and `compoundUnit` which together represent a Quantity in the form of

$$value \cdot 10^{power} compoundUnit.signature$$

See section 10 for details of the unit part and explanation of the term *signature* to understand how this framework operates.

To construct Quantities that are not included in this library, several steps need to be performed depending on the kind of the new signature:

- The simplest case: The new Quantity is just a data holder, and it will never be calculated with or converted into a similar Quantity. Then just inherit from `AbstractQuantity` and create constructors that take a `value` (and a `power`). Since every Quantity needs a *CompoundUnit* but in this case it does not matter which one, one can just use the `DimensionlessUnit` that can be created using `FrequentlyUsedUnits.getDimensionlessUnit()`. The constructor then looks like this:

  ```
  public DataHoldingQuantity(double value) {
     super(value, FrequentlyUsedUnits.getDimensionlessUnit());
     this.compoundUnit.setUnitName("newUnitName");
  }
  ```

  assuming the new Quantity should have a name called `newUnitName`.

- If the new unit is a combination of already existing units, like km^4/h, for example, and the new Quantity is to be converted or added to another (not yet existing) Quantity with the unit m^4/s – which is the *same* signature –, in this case 3 new classes are needed. The first class is the abstract superclass of the other two classes, it inherits the class

AbstractQuantity. By convention it should be called something like AbstractNewSignatureName. The constructors simply call the constructors of AbstractQuantity getting all the parameters they need to pass by, and the methods add and substract are made visible, but restricted to parameters of AbstractNewSignatureName. So the class would look like:

```
public abstract class AbstractNewSignatureName extends
        AbstractQuantity {

    public AbstractEnergy(double value, CompoundUnit unit) {
        super(value, unit);
    }

    public AbstractNewSignatureName(double value, int power,
            CompoundUnit unit) {
       super(value, power, unit);
    }

    public AbstractNewSignatureName(
            String recreationableString) {
        super(recreationableString);
    }

    public void add(AbstractNewSignatureName toAdd)
            throws WrongUnitException {
        super.add(toAdd);
    }

    public void substract(AbstractNewSignatureName toAdd)
            throws WrongUnitException {
        super.substract(toAdd);
    }

}
```

The specific usuable classes then are inherisons of the stated class. They need to specify a CompoundUnit, which is typically done by using a factory. See the implementation of FrequentlyUsedUnits for examples how this can be done. So, the code of the class would look like:

```
public class SpecialQuantity extends AbstractQuantity {

    public SpecialQuantity (double value) {
        super(value, SpecialQuantityFactory.getSpecialQuantity());
    }

    public SpecialQuantity(double value, int power) {
        super(value, power, SpecialQuantityFactory.getSpecialQuantity());
    }

    public SpecialQuantity(String recreationableString) {
        super(recreationableString);
    }
}
```

Sometimes it is useful to have some static methods that add or substract `AbstractNewQuantityName` objects returning a new object containing the result. See an arbitrary implementation in the library to learn how this could be done, i.e. `com.schneide.quantity.mechanicalQuantities.Meter`.

- If the designated new Quantity has a unit that has no appropriate `com.schneide.quantity.unit.Ato` in the library (like, for example, the first length of Meter, not the SI unit), a new `AtomarUnit` has to be created. See the documentation of `CompoundUnit` and `AtomarUnit` to learn more about that.

  Once the new `AtomarUnit` exists, the further work to get a working Quantity is described in step 2 above.

- If the new Quantity has a unit for which neither the `AtomarUnit` nor the `AtomarSignature` exist, for example a Quantity that counts apples, in addition to the new `AtomarUnit` a new `AtomarSignature` has to be created. You may wish to look at existing subclasses of `AtomarUnit` and `atomarSignature` to see how this is done.

10 The architecture of the units

A complete unit like $\frac{km^2}{s^2}$ is described in the class `CompoundUnit`. This class basically contains a map that contains `AtomarSignature` objects as key and `Unit` objects as value. Slightly different from the definition in the beginning, the class `Unit` describes the multiplicity of the `AtomarUnit` it contains.

A Unit knows its `AtomarUnit`, its multiplicity known as `int exponent` and its scale as `int scaleExponent`. The constants for the scale are defined in the class `ScaleUnit`.

An `AtomarUnit` has to know its `AtomarSignature`.

An `AtomarSignature` knows its reference unit, that is the unit other atomar units know the conversion factor to. In case of physical units, the reference unit is the SI unit. So, if someone wants to create an atomar unit describing english miles, the conversion factor to meter has to be defined in the new class `AtomarEnglishMile`.

11 Copyright