GET YOUR UNITS RIGHT

units of measurement

Java-Backend: JSR-363 / JSR-385

Frontend (ja, hier auch)

WHO AM I

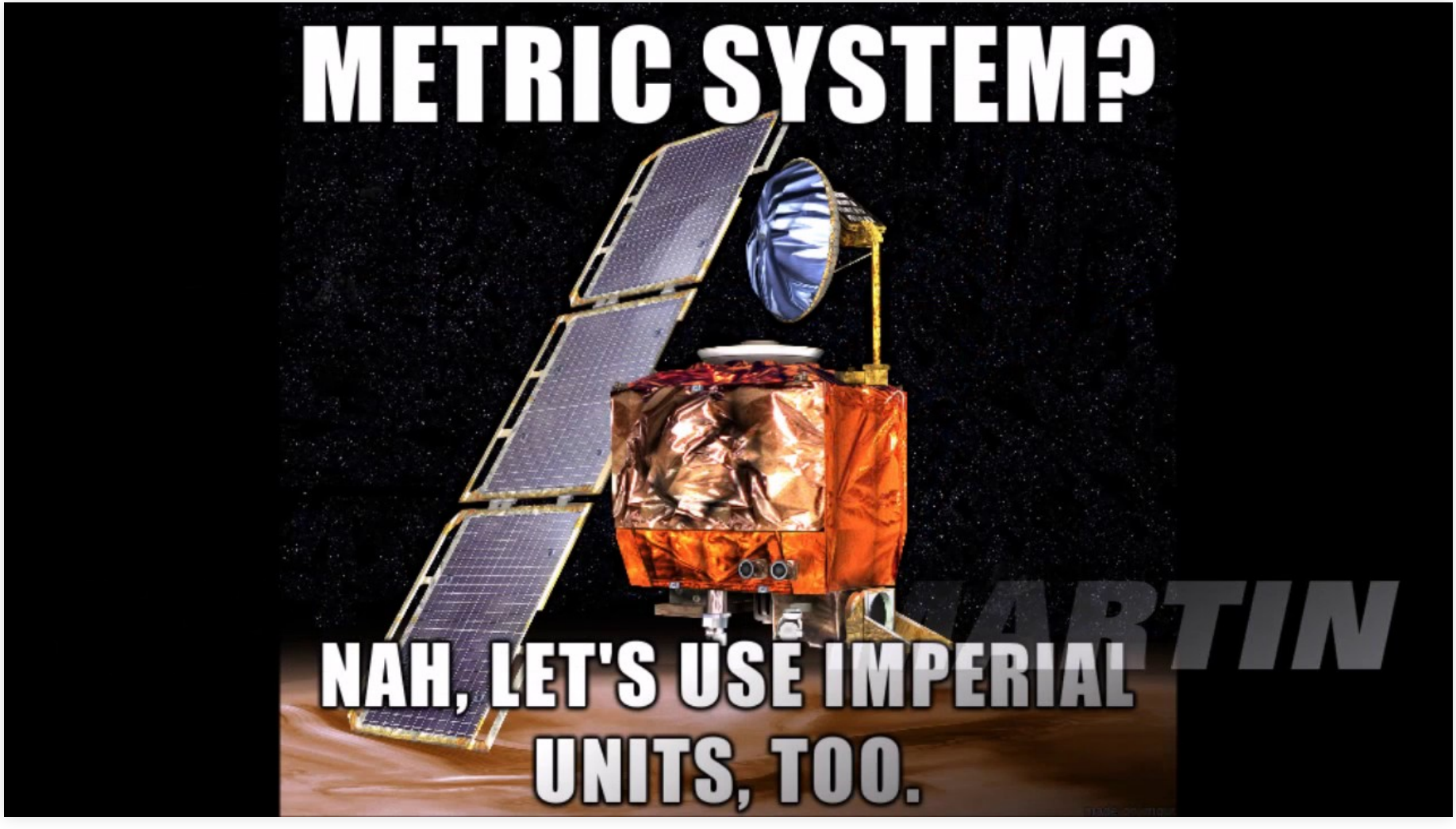
Felix Schultze

Consultant at @bridgingIT GmbH

Quarano e.V. supporter

EXAMPLE

1999: Mars Climate Orbiter => /dev/∞


METRIC SYSTEM?
NAH, LET'S USE IMPERIAL UNITS, TOO.

SYSTEM OF UNITS - REDEFINITION

In May 2019
All units are defined by constants of nature

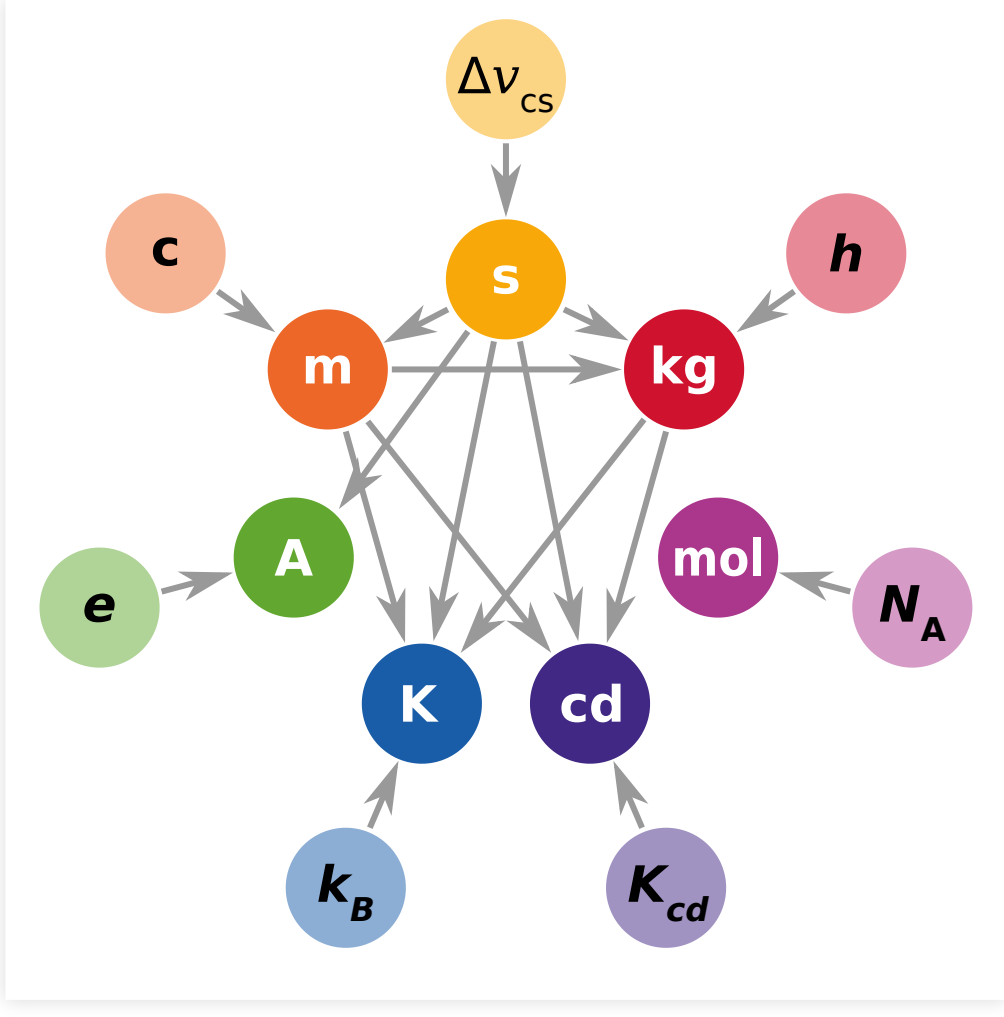
Konstante		exakter Wert	seit
$\Delta\nu_{Cs}$	Strahlung des Caesium-Atoms*	9 192 631 770 Hz	1967
c	Lichtgeschwindigkeit	299 792 458 m/s	1983
h	Plancksches Wirkungsquantum	$6,626\,070\,15 \cdot 10^{-34}$ J·s	2019
e	Elementarladung	$1,602\,176\,634 \cdot 10^{-19}$ C	2019
k_B	Boltzmann-Konstante	$1,380\,649 \cdot 10^{-23}$ J/K	2019
N_A	Avogadro-Konstante	$6,022\,140\,76 \cdot 10^{23}$ mol^{-1}	2019
K_{cd}	Photometrisches Strahlungsäquivalent**	683 lm/W	1979

* Hyperfeinstrukturübergang des Grundzustands des Caesium-133-Atoms

** für monochromatische Strahlung der Frequenz 540 THz (grünes Licht)

Source: https://de.wikipedia.org/wiki/Internationales_Einheitensystem

SYSTEM OF UNITS - REDEFINITION



Source: https://de.wikipedia.org/wiki/Internationales_Einheitensystem

SOME DEFINITIONS

Phrase	Example
Dimension	Time
Quantity symbol	t
Dimension symbol	T
Unit name	second
Unit symbol	s
Quantity	definite quantity (with/without dimension)

DISCLAIMER

Opinion ahead

RECOMMANDATION JAVAPRO

https://javapro.io/jsr-385-haette-mars-orbiter-retten-koennen/

JAVA CONSORTIUM

https://unitsofmeasurement.github.io/

JSR-363 -> API 1.0

JSR-385 -> API 2.x

Standard Implementations

MULTIPLE APIS

JAVAX.MEASURE 1.0

```
<dependency>
    <groupId>javax.measure</groupId>
    <artifactId>unit-api</artifactId>
    <version>1.0</version>
</dependency>
```

MULTIPLE APIS

JAVAX.MEASURE 1.0

```
<dependency>
    <groupId>javax.measure</groupId>
    <artifactId>unit-api</artifactId>
    <version>1.0</version>
</dependency>
```

JAVAX.MEASURE 2.X

```
<dependency>
    <groupId>javax.measure</groupId>
    <artifactId>unit-api</artifactId>
    <version>2.1.3</version>
</dependency>
```

STANDARD IMPLEMENTATION

```
<dependency>
    <groupId>tech.units</groupId>
    <artifactId>indriya</artifactId>
    <version>2.1.2</version>
</dependency>
```

JSR`S AMBITIONS

JSR`S AMBITIONS

- SI Units
  - Dimensions, Definitions, Faktors, Symbols, Scales

JSR`S AMBITIONS

- SI Units
  - Dimensions, Definitions, Faktors, Symbols, Scales
- Transformations

JSR`S AMBITIONS

- SI Units
  - Dimensions, Definitions, Faktors, Symbols, Scales
- Transformations
- Comparisons (in Standardimplementierung)

JSR`S AMBITIONS

- SI Units
  - Dimensions, Definitions, Faktors, Symbols, Scales
- Transformations
- Comparisons (in Standardimplementierung)
- extendable System of Units

JSR`S AMBITIONS

- SI Units
  - Dimensions, Definitions, Faktors, Symbols, Scales
- Transformations
- Comparisons (in Standardimplementierung)
- extendable System of Units
- Prefixes (Kilo, Milli, etc.)

JSR`S AMBITIONS

- SI Units
  - Dimensions, Definitions, Faktors, Symbols, Scales
- Transformations
- Comparisons (in Standardimplementierung)
- extendable System of Units
- Prefixes (Kilo, Milli, etc.)
- etc.

EXAMPLE // STANDARD SI QUANTITY

```
Quantity<Volume> cubicMetre

= Quantities.getQuantity(1, Units.CUBIC_METRE);
```

EXAMPLE // STANDARD SI QUANTITY

```
// Definition
Quantity<Length> distance
    = Quantities.getQuantity(10, MILLI(Units.METRE));

Quantity<Volume> cubicMetre
    = Quantities.getQuantity(1, Units.CUBIC_METRE);

Quantity<Volume> litres
    = Quantities.getQuantity(1000, Units.LITRE);
```

EXAMPLE // COMPARISONS IN INDRIYA

```
// Definition
ComparableQuantity<Volume> cubicMetre
    = Quantities.getQuantity(1, Units.CUBIC_METRE);

ComparableQuantity<Volume> litres
    = Quantities.getQuantity(1000, Units.LITRE);

// Comparison
assertTrue(cubicMetre.compareTo(litres) == 0);
```

EXAMPLE // OPERATIONS

```
Quantity<Volume> oneMoreLitre
    = cubicMetre.add(Quantities.getQuantity(1, Units.LITRE));

Quantity<Speed> velocity
    = Quantities.getQuantity(1, Units.METRE)
            .divide(Quantities.getQuantity(1, Units.SECOND))
            .asType(Speed.class);
```

EXAMPLE // DEFINITION OF NEW UNIT

EXAMPLE // DEFINITION OF NEW UNIT

```
Unit<Area> wrongSquareMM
    = MILLI(Units.SQUARE_METRE));
// toString(): mm² => results in milli * (m)²
```

EXAMPLE // DEFINITION OF NEW UNIT

```
Unit<Area> wrongSquareMM
    = MILLI(Units.SQUARE_METRE));
// toString(): mm² => results in milli * (m)²
```

```
Unit<Area> correctSquareMM
    = new ProductUnit<>(MILLI(Units.METRE)
        .multiply(MILLI(Units.METRE)))
        .asType(Area.class);
// toString(): mm² => results in (mm)²

Unit<Area> anotherCorrectSquareMM
    = ProductUnit.ofPow(MILLI(Units.METRE), 2)
        .asType(Area.class);
// toString(): mm² => results in (mm)²
```

ATTENTION //

DIFFERENT UNIT BASE CLASSES

Can be confusion in the beginning

- AlternateUnit
- AnnotatedUnit
- ProductUnit
- TransformedUnit

ATTENTION //

ALTERNATEUNIT

- Can be derived by multiple other units
- No scaling (Milli, etc.)

JavaDoc:

```
Unit<Force> NEWTON
    = AlternateUnit.of(
        METRE.multiply(KILOGRAM).divide(SECOND.pow(2)),
        "N"
      )
    .asType(Force.class);
```

ATTENTION // ANNOTATEDUNIT

- toString(): unit{annotation}

```
Unit<Length> ANNOTATED
    = AnnotatedUnit.of(Units.METRE, "annotation");
// => m{annotation}
```

Attention //

ProductUnit

- Can be derived by multiple other units
- Scaling possible (Milli, etc.)
- No definition of unit symbol

```
Unit<Area> SQUARE_MILLI_METRE
    = new ProductUnit<>(
        MILLI(Units.METRE).multiply(MILLI(Units.METRE))
    ).asType(Area.class);
```

ATTENTION // TRANSFORMEDUNIT

- "Transformed" by other units
- "Transformed" with converters
- No definition of unit symbol, just "label"

```
Unit<Pressure> BAR
    = new TransformedUnit<>(
        "bar",
        Units.PASCAL,
        MultiplyConverter.of(100_000)
      );
```

RECOMMENDATION

Keep your units in a single place

```
class Units {
    public static final Unit<Area> SQUARE_MILLI_METRE = ...;
    public static final Unit<Area> BAR = ...;
}
```

RECOMMENDATION

Encapsulate your own Quantites by delegation

```
class PipeCrossSection extends ... {

    private PipeCrossSection(ComparableQuantity<Area> q) {
        super(q);
    }

    static PipeCrossSection of(Quantity<Area> quantity) {
        ComparableQuantity<Area> to = quantity.to(SQUARE_MILLI
        return new CableCrossSection(to);
    }
}
```

RECOMMENDATION

In indriya:2.1.2, its not that easy anymore....

```
abstract class BaseQuantity<Q extends Quantity<Q>> extends Abs
    private ComparableQuantity<Q> delegate;

    protected BaseQuantity(ComparableQuantity<Q> delegate) {
        super(delegate.getUnit());
        this.delegate = delegate;
    }

    @Override
    public Number getValue() {
        return delegate.getValue();
    }

    @Override
    public ComparableQuantity<Q> add(Quantity<Q> quantity) {
```

JPA

1st idea: save concrete string

JPA

1st idea: save concrete string
DB: VARCHAR => e.g.: "10 mm²"

JPA

2nd idea: save string in base unit

JPA

2nd idea: save string in base unit
DB: VARCHAR => e.g.: "0.0001 m²"

```java
@Entity
class Pipe {
    @Column(name = "CROSS_SECTION")
    @Convert(converter = CrossSectionConverter.class)
    PipeCrossSection crossSection;
}
```

```
@javax.persistence.Converter
class CrossSectionConverter
    implements AttributeConverter<PipeCrossSection, String> {

    @Override
    String convertToDatabaseColumn(PipeCrossSection qs) {
        return qs.to(Units.SQUARE_METRE).toString();
    }
    @Override
    PipeCrossSection convertToEntityAttribute(String dbData) {
        Quantity<Area> area = Quantities.getQuantity(dbData)
                    .asType(Area.class);

        return PipeCrossSection.of(area);
    }
```

JPA

3rd idea: save plain number

JPA

3rd idea: save plain number
DB: NUMBER => e.g.: "0.0001"

```
@Entity
class Pipe {
    @Column(name = "CROSS_SECTION")
    @Convert(converter = PipeCrossSectionConverter.class)
    PipeCrossSection crossSection;
}
```

```
@javax.persistence.Converter
class PipeCrossSectionConverter implements AttributeConverter<
    @Override
    public String convertToDatabaseColumn(PipeCrossSection qs)
        return qs.to(Units.SQUARE_METRE).getValue();
    }

    @Override
    public PipeCrossSection convertToEntityAttribute(Number db
        Quantity<Area> area = new AreaQuantity(dbData, Units.S
        return PipeCrossSection.of(area);
    }
}
```

JPA

4th idea: usage of an embedded class

JPA

4th idea: usage of an embedded class
DB: NUMBER => 0.0001 and UNIT => 'mm²'

```
@Entity
class Pipe {
    @Embedded
    PipeCrossSection crossSection;
}
```

```java
@Embeddable
class PipeCrossSection {
    @Column(name = "cross_section_value")
    private BigDecimal value = BigDecimal.ZERO;
    @Column(name = "cross_section_unit")
    private String unit = SQUARE_MILLI_METRE.toString();

    @Transient
    ComparableQuantity<Area> getCrossSection() {
        Unit<Area> parse = SimpleUnitFormat.getInstance()
            .parse(unit)
            .asType(Area.class);
        return Quantities.getQuantity(value, parse);
    }
```

hmm, kay,
what about frontend integration?!?!?!?!

ANGULAR / FRONTEND

ng-units:
https://hansmaad.github.io/ng-units/

ANGULAR / FRONTEND

ng-units:
https://hansmaad.github.io/ng-units/

- Pretty good for just showing quantities
- SystemOfUnits not that well done
- unintuitive usage

FRONTEND

js-quantities (js/ts):
https://github.com/gentooboontoo/js-quantities

FRONTEND

js-quantities (js/ts):
https://github.com/gentooboontoo/js-quantities

- Free definition of new/own units
- Transformations possible
- Formatting possible
- etc...

INSTALLATION

```
npm install js-quantities --save
npm install @types/js-quantities --save # type definitions
```

IMPORTS

```
import * as Qty from 'js-quantities';
```

SOME BASICS

```
// Creation
new Qty('1 m') // 1 meter
new Qty('1 m^2') // 1 m²
new Qty('1 m2') // 1 m²

// Transformation
new Qty('1 m2').to('mm2') // 1000000 mm²

// Formatting
new Qty('1.23 m').toPrec(0.1) // 1.2 m
```

INTEGRATION NG

Idea: Usage of plain *Qtys*
and rendering with pipes

INTEGRATION NG

Idea: Usage of plain *Qtys* and rendering with pipes

Pipe

```
export class QtyPipe implements PipeTransform {

  transform(value: Qty, ...args: unknown[]): unknown {
    return value.toString();
  }

}
```

Template

```
Pressure: {{ pipe.pressure | qty }} => 5 bar
```

INTEGRATION NG

Defined your own unit?

INTEGRATION NG

Defined your own unit?

Pipe

```
export class CrossSectionPipe implements PipeTransform {

  transform(value: Qty, ...args: unknown[]): unknown {
    return value.toString().replace(/2$/, '²');
  }

}
```

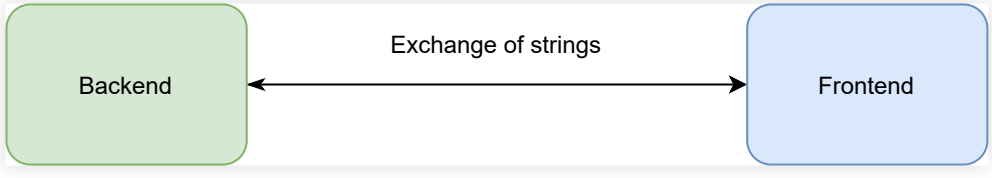
Template

```
Cross-Section: {{ pipe.crossSection | crossSection }} => 10 mm
```

ROUNDUP OF JS-QUANTITIES

- parsing of many units by default
- parsing of 'mm²' fails, easy to fix
- 'bar', which has to be defined in backend
  - exists in frontend

INTEGRATION FRONTEND / BACKEND




```
"pipe": {
    "pressure": "10 bar",
    "crossSection": "10 mm²"
}
```

SUMMARY

- Units keep your code type-safe
- Units define your quantites explicitly
- Dokumentations (FE/BE) can be improved ;-)
- Integration FE<->BE pretty easy
- If you do not need transformations, etc
  - => maybe a plain ValueObject is enough :-)

SOURCES

1. https://unitsofmeasurement.github.io/
2. https://de.wikipedia.org/wiki/Internationales_Einheitensystem#Seit_2019:_Definition
3. https://github.com/hansmaad/ng-units
4. https://github.com/gentooboontoo/js-quantities
5. https://de.wikipedia.org/wiki/Liste_von_Programmfehlerbeispielen
6. https://www.astronews.com/news/artikel/1999/10/9910-001.shtml
7. https://javapro.io/jsr-385-haette-mars-orbiter-retten-koennen/