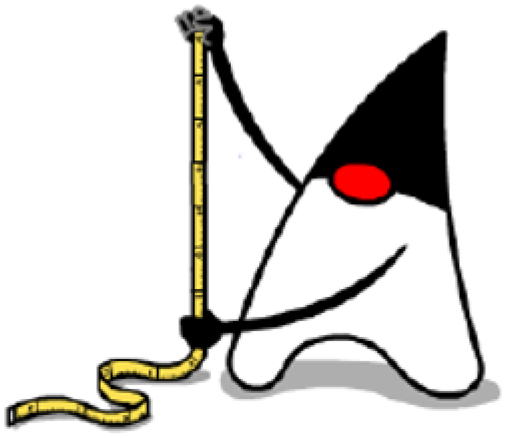


JSR 385

Units of Measurement

Version 2.2 – October 31, 2023

LICENCE

Introduction

Objective

A framework supporting robust representation and correct handling of quantities is a basic need of Java developers across domains including science, engineering, medicine, finance or manufacturing. Currently, developers have to either use an inadequate model of measurement or to create a custom solution. Either choice can lead to significant programmatic errors. The default practice of modeling a measure as a simple number with no regard to the units it represents creates fragile code, as the value may easily be misinterpreted if the unit must be inferred solely from contextual clues. For example, it may be unclear whether a person's mass is expressed in pounds, kilograms, or stones.

A standard solution is both safer and more efficient, saving developers' valuable time for domain-specific work. JSR 385 proposes to establish safe and useful methods for modeling physical quantities. The specification includes:

- Interfaces and abstract classes with methods supporting unit operations, including:
    - Checking of unit compatibility
    - Expression of measurement in various units
    - Arithmetic operations on units
- Concrete classes implementing standard unit types (such as base and derived) and unit conversions.
- Concrete classes for parsing and formatting textual unit representations.
- A “database” or equivalent repository/system of predefined units.

Expert Group

This work is being conducted as part of JSR 385 under the Java Community Process Program.
This specification is the result of the collaborative work by members of the JSR 385 Expert Group and the community at large. The following people are EG members of JSR 385:

- Mohammed Al-Moayed
- Teo Bais
- Jean-Marie Dautelle
- Daniel Dias dos Santos
- Martin Desruisseaux - Geomatys
- Jacob Glickman
- Magesh Kasthuri
- Werner Keil
- Otávio Santana

Contributors

JSR 385 applies version 2.11 of the Java Community Process. Which defines a JSR member category of Contributor. The following people are contributors to JSR 385 in alphabetical order:

- Ellis Berry
- Mads Opheim - Computas
- Andi Huber
- Rustam Mehmandarov
- Anakar Parida
- Matthijs Thoolen - Utrecht Java User Group
- Filip van Laenen

1. Requirements and Design Goals

Requirements

Core API requirements include the following:

- No change to the Java Virtual Machine.
- Do no numerical harm. The unit framework should in no way impact business logic.
- Arithmetic operations supported by the framework should as much as possible be aligned with existing arithmetic operations in the Java Virtual Machine, e.g. the BigDecimal class both in terms of naming and behavior.
- Interoperability with code that does not use the unit framework defined in the API.
- Support for user-defined dimensions, units, quantities and systems of units.
- Support for strict symbol parsing, including:
    - Metric prefixes mainly for use with Système International (SI) units.
    - Other prefixes (e.g. locale or culture specific) with all units, but with contextually limited parsing and formatting (they may be displayed simply as *"unit*N"* in another context).
- Support for runtime handling and printing of units, quantities, and dimensions.

For example, a unit that is ($kg^1 \cdot m^1 \cdot s^{-2}$) should be able to expose the dimensions (kg, m, s), the exponents of those dimensions (1, 1, -2), and/or the name of the unit (if there is a defined name) at runtime.

Aspirations

The following features are considered important to the success of the API:

- Should provide those currently working with numeric types extending `Number` a low enough learning curve to perform these calculations in a unit consistent manner.
- Support for the use of more than one unit system simultaneously. For example, a program should be able to simultaneously operate on SI, centimetre-gram-second (CGS), and imperial units.
- Compile-time detection of dimension errors when sufficient information is available for a compile-time check.

Source Code and Binary Compatibility

Java 8 source code and compiler compatibility is used for the API. The JAR is Java 8 compatible, but it also contains a Java 9 compliant **module-info**. While the Oracle Support Roadmap for Java SE[1] shows Java 7 is still supported until 2022 (longer for Sustaining Support customers) users of either Java SE 7 or Java ME 8 Embedded shall continue using [JSR-363] until their environment or new applications are ready to use JSR 385.

Although a dedicated Java SE Embedded no longer exists after version 8, it runs on many devices including small ones like the Raspberry Pi Zero. Using JLink from Java 9 on the size of the JVM and modules needed for a particular application can be further reduced, which makes the decision to drop a dedicated Java SE Embedded and also Java ME Embedded after version 8 understandable.

For the reference implementation (RI) the minimum Java version is Java SE 8.

[1] See Oracle Support Roadmap for Java: http://www.oracle.com/technetwork/java/eol-135779.htm

2. Definitions

The measurement of a physical quantity is an estimation of its magnitude in relationship to a well-known quantity of the same kind, which we take for unity. For example, “5 metres” is an estimate of an object's length relative to another length, the metre, which we adopt in this example[2] as the standard unit of length. A similar approach can be taken for monetary quantities[3], providing that a monetary unit is properly defined.

This definition assumes a zero-based linear scale. However, measurements can be *expressed* using different scales for convenience. For example, a measurement in Degrees Celsius can be thought of as a measurement in Degrees Kelvin offset by 273.15 degrees for convenience. A measurement in decibel can be thought of as a power measurement expressed on a logarithmic scale.

The term “measurement” is also used in a looser fashion, to refer to the arbitrary assignment of numbers to represent scale. For example, the Mohs scale of mineral hardness[4] characterizes the scratch resistance of various minerals according to the ability of a harder mineral to scratch a softer mineral. This scale indicates that quartz is harder than calcite, but not how much harder. It is a strictly *ordinal* scale. Quantities expressed according to such scales are comparable but not additive.

A measurement can also be an exact quantity. For example, we can determine that there are exactly 12 eggs in a carton by counting them.

Symbols and abbreviated terms

API	Application Programming Interface
BIPM	Bureau International des Poids et Mesures
CGPM	General Conference on Weights and Measures
CGS	Centimetre-gram-second system of units
CLDC	Connected Limited Device Configuration
ISO	International Organization for Standardization
Java SE	Java Standard Edition
Java ME	Java Micro Edition
JSR	Java Specification Request
NIST	National Institute of Standards and Technology
RI	Reference Implementation
SI	Système International
SPI	Service Provider Interface
TCK	Technology Compatibility Kit
UCUM	Unified Code for Units of Measure
UOM	Units of Measurement
UML	Unified Modeling Language

Formatting

Aside from normal plain text:

- *Italic* is used to emphasize terms and keywords;
- **Bold** is used mainly for important paragraphs, notes or remarks;
- `Serif` is used for pieces of code, class or interface names, as well as entire code samples;
- Links point to cross references inside the document or external URLs.

[2] This specification does not mandate the use of any particular quantity as a standard unit. However, we expect the Système International (SI) to be the standard system of units for most applications.

[3] Monetary quantities is the topic of JSR 354, Money and Currency API.

[4] http://en.wikipedia.org/wiki/Mohs_scale_of_mineral_hardness

Terms used in this specification

Where feasible, the full title of this JSR “Units of Measurement API” (for Java) is used by this document, or “JSR 385”. A short form like “Unit-API” generally stands for the API part, that may also simply be called “API”.

The Java classes defined in this specification are not limited in application to physical quantities. The same API can be applied to monetary quantities, (Agile) story points, clothing sizes or to user-defined scales, including ordinal scales. Consequently, the terms “*quantity*” and “*unit*” should be taken in their general sense unless otherwise specified, as in “*physical quantity*” or “*physical unit*”.

Dimension

One of the fundamental aspects of quantities, such as length (L), mass (M), time (T), or a combination thereof (ML/T^2, the dimension of force).

Dimension expresses a property without any concept of magnitude. We can talk about length (L) without any implication that we are talking about any particular length. Two or more quantities are said to be commensurable if they have the same dimensions. Quantities that are commensurable can be meaningfully compared.

Quantity

Any type of quantitative property or attribute of a thing. For example, temperature, volume, pressure, molecular mass and internal energy are quantitative properties which can be used to describe the state of a confined gas.

A unit is not needed to express a quantity. For example, Alice can quantify the mass of her shoe by handling it. A unit is not needed to do quantitative arithmetic, either. Alice can add the mass of her left shoe and the mass of her right shoe by placing them both in the pan of a balance. Units are needed to represent *measurable* quantities in a computer, on paper, on a network, *etc.*

In this API, quantities are restricted to the measurable ones: only the quantities that can be expressed as the combination of a numerical value and a unit are supported. This is sometimes considered as a partial[5] definition of *measure* rather than *quantity*. Nevertheless this API uses the “quantity” term for consistency with usage in other frameworks, because the concept of quantity is needed anyway for unit parameterization, and for avoiding the introduction of both concepts in the library.

Unit

A quantity adopted as a standard, in terms of which the magnitude of other quantities of the same dimension can be stated.

Units are created from some well-known quantity taken as a reference. Regardless of how it is created, a unit can be expressed as a quantity of other units with the same dimension. For example, the foot unit corresponds to a quantity of 0.3048 metres.

Base unit

A well-defined unit which by convention is regarded as dimensionally independent of other base units. The SI system defines only 7 base units (including *metre*, *kilogram* and *second*), from which all other SI units are derived.

For example in the imperial system, only *one* of “inch” or “foot” can be selected as a base unit. All other units in the same dimension are defined as the selected base unit scaled by some factor.

[5] A more complete definition would include information about the precision of the measurement.

Derived unit

A unit formed as a product of powers of the base units. For example, the square metre (m^2) is a unit derived from the metre (m).

Some derived units get a special name and symbol for convenience. For example the SI unit of power ($m^2 \cdot kg \cdot s^{-3}$) is named *watt* and is represented by the symbol W. Other derived units with special name include *hertz*, *newton* and *volt*.

In the SI, the base and derived units form a coherent set of units, where coherent means that those units are mutually related by rules of multiplication and division with no numerical factor other than 1 (BIPM, 1998). Despite this definition, the term "derived unit" is sometimes used in a looser fashion for the result of a unit scaled to some factor (for example, kilometre defined as 1000 metres, also see Equivalence) or transformed in any other way (logarithmic, offset).

System of units

A set of base and derived units chosen as standards for specifying measurements. Examples include the SI (Metric), US or Imperial System.

Different systems of units may share the same units. For example, the US system contains many of the units already defined in the Imperial System (e.g. foot).

Prefix

A leading word that can be applied to a unit to form a decimal multiple or submultiple of the unit. Prefixes are primarily used in the SI system, for example *kilo-* and *centi-*, but other common prefixes like *binary* (used especially in information technologies) are also provided.

Unit converter

A converter of numeric values between commensurable units.

Unit format

Formats instances of `Unit` to a `String` or an `Appendable` and parses a `CharSequence` (`String`) to a `Unit`.

Quantity format

Formats instances of `Quantity` to a `String` or an `Appendable` and parses a `CharSequence` (`String`) to a `Quantity`.

Metre or Meter?

The former is British spelling and the latter is U.S. The BIPM brochure, ISO 31 and Wikipedia use “metre”, while NIST uses “meter” (e.g. NIST SP 1038). For this specification, we retained the BIPM “metre” spelling in the text unless explicitly required otherwise. All SI constants names (e.g. `Units.METRE`) use the BIPM spelling.

The two spellings are sometimes used together with slightly different meanings: “metre” as a measure and “meter” as a measuring device.

Equivalence

Strict equality in Java makes a difference between `Integer.valueOf(1)`, `BigInteger.ONE` or `BigDecimal.ONE` despite the numeric value being the same. That’s why we introduced equivalence. For example kilometre is a `Unit` of `Length` in the metric system **equivalent to** 1000 metres. 1 km is also equivalent to 0.6214 miles.

Example

The result of an experience measuring the wavelength of monochromatic light emission may be expressed in the SI system of units as:

$$\lambda = 698.2 \text{ nm}$$

where:	λ	is the symbol for the physical quantity (wavelength),
	nm	is the symbol for the physical unit (nanometre), where:
	n	is the symbol for the sub-multiple (nano, meaning 10^{-9}),
	m	is the symbol for the base or derived unit on which the prefix is applied.
	698.2	is the numerical value (magnitude) of the wavelength in nanometres.

The dimension of length is typically represented by the letter L in dimensional analysis. However, this dimension does not appear explicitly. Instead, it can be inferred from the commensurable quantity “λ”, or from the unit “m”.

3. Use Cases

E-Commerce

Online-Shop

One basic scenario that must be covered is a traditional web shop. Hereby products are presented and collected in a shopping cart. Each product can be added once or multiple times to the cart. Some sites also need to represent non-integral amounts, such as 1.5 kg of a product. Additionally a site may be internationalized handling multiple unit systems, perhaps controlled by user settings or address. Summarizing this scenario implies the following requirements:

- The user may change the number of each item to purchase, either by defining an integral number (e.g. 2 products) or a decimal point number (e.g. 1.5 kg). This requires multiplication with integer and decimal numbers.
- Users from different countries require different units, e.g. 1.5 kg in most countries where the metric system applies, but 3.31 lb in others, especially the United States.

Internet of Things

The Internet of Things holds a vast number of use cases for units and quantities. These are just a few.

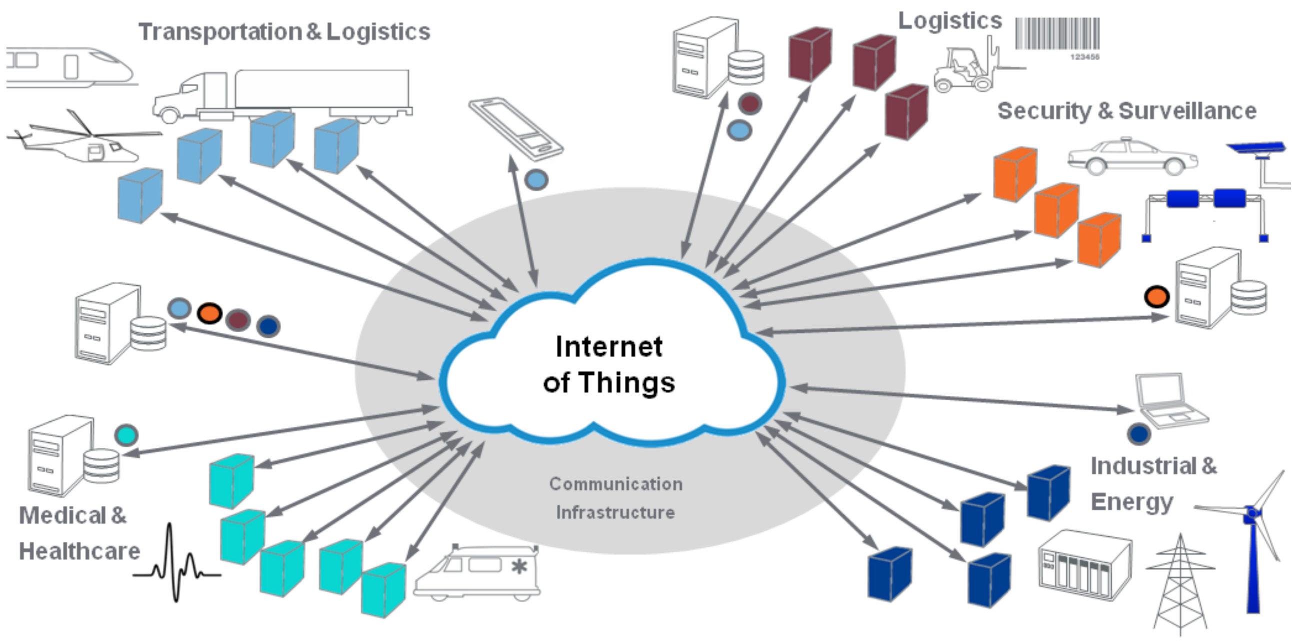


Image © 2012 Eclipse Foundation, Inc. Made available under the Eclipse Public License 1.0

An example use case is a connected gas pump, that handles information about the volume of combustible (be it gasoline, alcohol or liquefied petroleum gas) sold, price information and current tank volume. Having these values in their correct units prevent wrong price calculation, make it easier to calculate effects like volume expansion caused by heat and transfer and show the value information on multicultural displays.

Medical & Healthcare

Healthcare providers and organizations around the world made the Unified Code for Units of Measure [UCUM] a widely used standard for their communication, especially HL7 or large globally operating institutions like WHO or CDC who are in need of sustainable and reliable exchange for healthcare and patient information.

Quantified Self

The Personal Health and Fitness trend commonly referred to as "Quantified Self" already contains the term *Quantity*. Personal health devices range from "Smart Pill Boxes" (knowing the dosage a patient needs based on their EHR) to connected scales, blood pressure, heartbeat or sugar sensors.

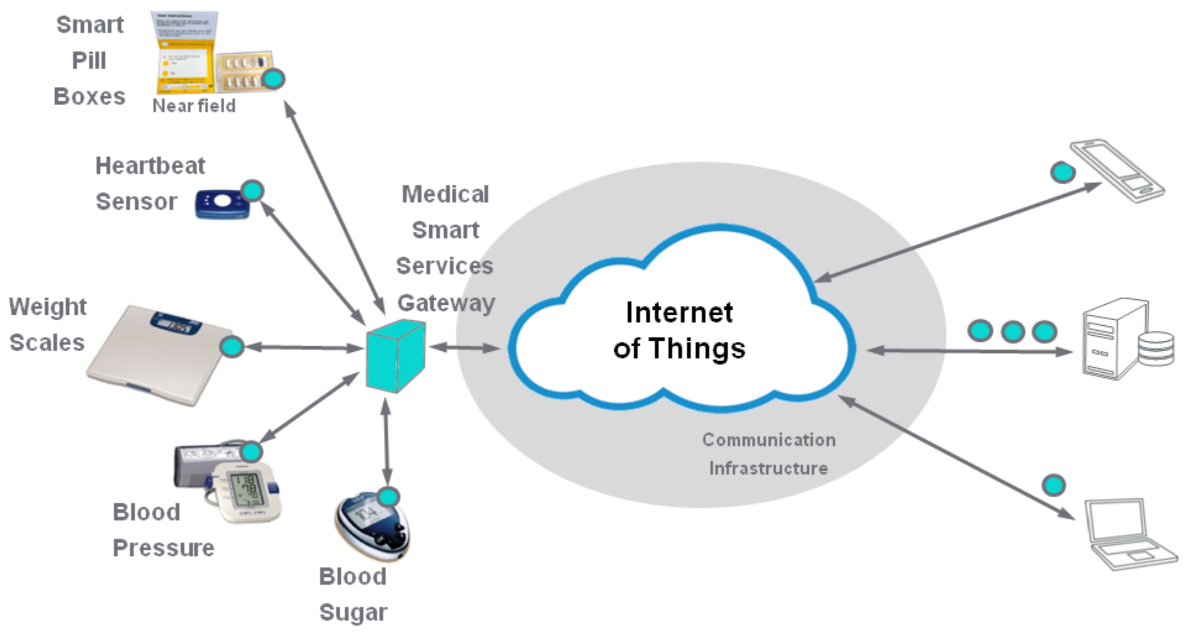


Image © 2012 Eclipse Foundation, Inc. Made available under the Eclipse Public License 1.0

A connected scale knowing the person's height can easily calculate the current Body Mass Index.

DevOps & Cloud

DevOps, Configuration Management or The Cloud rely on measurement everywhere. Without measuring available capacity, bandwidth and other vital stats of service or how much users already consumed Cloud based solutions would be unthinkable or at least inefficient, error-prone and lossy.

Monitoring

Monitoring Cloud solutions, often performance-related is a classical use case where measurements and their units play an important role.

4. API

Packages and fundamental types

This specification consists of the following packages:

- `javax.measure`
  - `javax.measure.format` [OPTIONAL] [6]
  - `javax.measure.quantity` [OPTIONAL]
  - `javax.measure.spi` [OPTIONAL]

The main types of this API, namely `Dimension`, `Quantity` and `Unit`, are defined in the top level `javax.measure` package.

The `javax.measure.quantity` package defines quantity types (e.g. `Mass`, `Length`), while the `javax.measure` package defines the units in which quantities can be expressed (e.g. gram, metre). Quantities and units work in synergy through the use of Java Generics.

Note: some examples provided in this API chapter contain upper-case identifiers like `METRE`. These identifiers stand for static members of the `Units` class available in the Reference Implementation. Other implementations may provide equivalent members. These members are generally omitted from discussion since all of them can be implemented in terms of calls to `Unit` member methods.

The Unit Interface

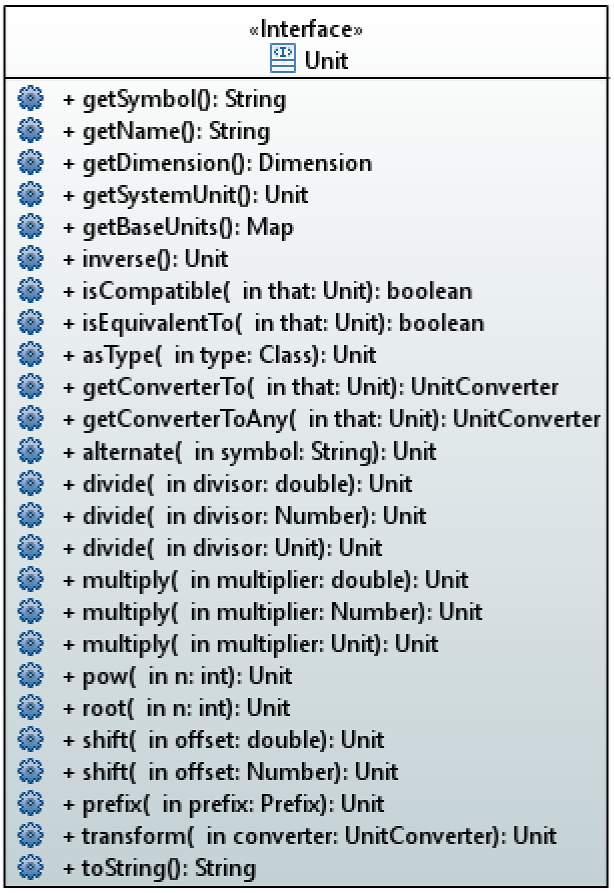
Probably the most fundamental interface with which the user needs to be familiar is `javax.measure.`**`Unit`**. The unit type represents a well-known quantity that has been adopted as a standard of measurement. For example "*kilometre*" and "*watt*" are units.

Units and quantities are defined in terms of each other, but are distinct. "*5* kilometres" is a quantity, but is not a unit. While `Unit` could have been defined as a subtype of `Quantity`, in the sense of "*is a kind of*", this specification avoids creating this relationship in order to discourage the abuse of `Unit` as a general-purpose implementation of `Quantity`.

`Quantity` and `Unit` are both parameterized by a quantity type (Unit Parameterization).

In addition to the `equals()` and `hashCode()` method of every object, `Unit` also supports equivalence via the `isEquivalentTo()` method. Unlike `isCompatible()` an equivalence check requires both units to be strictly type-compatible (see Unit Parameterization), because it makes no sense to compare e.g. gram and mm for equivalence. While the compatibility check may also works across different quantity types.

[6] See Optionality


«Interface»
Unit
+ getSymbol(): String
+ getName(): String
+ getDimension(): Dimension
+ getSystemUnit(): Unit
+ getBaseUnits(): Map
+ inverse(): Unit
+ isCompatible(in that: Unit): boolean
+ isEquivalentTo(in that: Unit): boolean
+ asType(in type: Class): Unit
+ getConverterTo(in that: Unit): UnitConverter
+ getConverterToAny(in that: Unit): UnitConverter
+ alternate(in symbol: String): Unit
+ divide(in divisor: double): Unit
+ divide(in divisor: Number): Unit
+ divide(in divisor: Unit): Unit
+ multiply(in multiplier: double): Unit
+ multiply(in multiplier: Number): Unit
+ multiply(in multiplier: Unit): Unit
+ pow(in n: int): Unit
+ root(in n: int): Unit
+ shift(in offset: double): Unit
+ shift(in offset: Number): Unit
+ prefix(in prefix: Prefix): Unit
+ transform(in converter: UnitConverter): Unit
+ toString(): String

The dimensions of a unit (`javax.measure.`**`Dimension`**) can be retrieved at runtime from the unit interface. At compile-time, the dimensions must be inferred from its quantity type.

> **Example:** "centimetre" is a unit. "*2.54* centimetres" is a quantity of type `Length`. This quantity can also be used as the definition of a new length unit called "*inch*". Note that "*1 inch*" (the quantity) and "*inch*" (the unit) are not synonymous. In this API, they are represented by two distinct types.

All numerical values that result from a measurement are associated, directly or indirectly, to a unit. The simplest quantity is the combination of a numerical value with a unit. Such quantities can be created directly through implementations of the `QuantityFactory` interface.

The association of many numerical values to the same unit can be aggregated into container objects such as vectors, columns in a table etc. Such constructs outline a means of storing homogeneous measurements.

For example:

```
public class Vector<Q extends Quantity<Q>> {
   double[] values;
   Unit<Q> unit; // Single unit for all vector components.
   ...
}
```

Associating a single unit with a large set of numerical values is not only more efficient in terms of storage space, it is also more performant since the formula to convert the values to another unit must be determined only once for a whole vector. A single unit can also be associated with more complex objects, like matrices or collections.

Since the same `Unit` instances are typically referenced by a large number of quantity objects, immutability is essential as the cost of cloning each unit (*defensively copying*) would be prohibitive. `Unit` instances are immutable and thus thread-safe. They are safe for use as final static constants.

Operations on Units

Units can be created dynamically as the result of algebraic operations on existing unit objects. All operations are defined as member methods of the `Unit` interface.

Operands

Most binary operations expect (`Unit`, `Unit`) or (`Unit`, *n*) operands where *n* stands for a real value (sometimes restricted to an integer value). The first operand is always the `Unit` instance on which the Java method is invoked.

> **Example:** let “m” and “s” be two instances of `Unit`. Some valid operations are “m/s”, “m·s”, “m^2” or “m·1000”, where the operations represented by “/”, “·” and raising to a power corresponds to the Java methods `divide(double)`, `multiply(double)` and `pow(int)` invoked on the `Unit` instance named “m”.

Result

All operations – both unary and binary – return a new or an existing instance of `Unit`. Operations can be categorized by comparing the dimension of the returned unit to the dimension of the first `Unit` operand:

Same dimension as the operand

These operations define new units by either scaling existing units by some factor, or by translating existing units by some offset. Quantities expressed in terms of the resulting unit are convertible to the original unit. For example, the inch unit can be defined as 2.54 centimetres, or equivalently as the centimetre unit scaled by a factor of 2.54.

The following examples define the inch unit as 2.54 centimetres. This unit is defined in three different ways, which are all equivalent up to some rounding error. Note that `CENTI(x)` is a convenience method for `x.divide(100)` with prefix symbol management added. The `CENTI` and `METRE` members are defined in the `Units` class defined in the reference implementation, which is assumed implicitly through a static `import` statement. The result is a new unit (inch) of the same dimension (length) as the scaled unit (metre).

```
Unit<Length> INCH = CENTI(METRE).multiply(2.54);
Unit<Length> INCH = METRE.multiply(0.01).multiply(2.54);
Unit<Length> INCH = METRE.multiply(254).divide(100); // Recommended.
```

The last definition (using only integer values) is recommended as it does not introduce floating point imprecision (the internal representation of 2.54 using the double primitive type is actually something like 2.540000000000000036…).

Different dimension than the operand

These operations are used for derivation of new units from existing ones (often from base units). Quantities expressed in terms of the resulting unit are usually not convertible to the original unit. For example, the *watt* unit can be defined as the *joule* unit divided by the *second* unit.

The following example defines the watt unit as the joule unit divided by the second unit. Joule has the dimension of energy, while second has the dimension of time. The result is a unit with dimension of power.

```
Unit<Power> WATT = JOULE.divide(SECOND).asType(Power.class);
```

The following table summarizes the algebraic operations provided by the Unit interface.

Result with same dimension	***Result with different dimension***
Binary operations: **divide** (double) **divide** (Number) **multiply** (double) **multiply** (Number) **shift** (double) **shift** (Number) **alternate** (String) **prefix** (Prefix)	*Binary operations:* **pow** (int) **root** (int) **multiply** (Unit) **divide** (Unit) **transform** (UnitConverter)
	Unary operations: **inverse**()

Unit Parameterization

Units are always checked for compatibility at runtime prior to any operation. For example, any attempt to convert a quantity from *kilogram* units to *metre* units will result in a `MeasurementException` being thrown, although in most cases the **compiler** wouldn't even allow most of these wrong conversions. Rigorous runtime checks are needed because in some cases units may be unknown at compile-time, and because it is possible to defeat the compile-time checks with unchecked casts. Note that the performance impact of systematic runtime checks is not always significant.

In addition to runtime checks, some limited compile-time checks can be achieved in the Java language using parameterized types. Units can be parameterized with the appropriate quantity type for a type parameter. For example, the `Unit` interface should be parameterized as `Unit<Length>` for any units of length to restrict the units they can accept.

There are two ways to obtain a parameterized unit:

- Assignment from one of the predefined constants, such as those defined in the `Units` class.
- The return value from any operation returning a unit of the same type as the operand – or, in terms of Java language, all `Unit` methods where the return type is exactly `Unit<Q>`. This includes multiplication and division by a dimensionless factor.

The following assignments are examples of type safe expressions:

```
Unit<Length> m    = METRE;
Unit<Length> cm   = CENTI(m);
Unit<Length> inch = cm.multiply(2.54);
```

However, because the result is a type that cannot be determined statically by the Java type system the assignments below are not type safe, and require an explicit cast to avoid a compiler error. The Java compiler emits an “unchecked cast” warning for such code.

```
Unit<Length>   m  = (Unit<Length>)   AbstractUnit.parse("m");   // Unsafe!
Unit<Area>     m2 = (Unit<Area>)     m.pow(2);                  // Unsafe!
Unit<Pressure> Pa = (Unit<Pressure>) NEWTON.divide(m2);         // Unsafe!
```

Note, `AbstractUnit` is an implementation of `Unit` in the RI, mentioned here as an example.

As of Java 8, checks cannot be performed at compile time for such code. However, the above code can be rewritten in a slightly safer way as follows:

```
Unit<Length>   m  = AbstractUnit.parse("m").asType(Length.class);
Unit<Area>     m2 = m.pow(2).asType(Area.class);
Unit<Pressure> Pa = NEWTON.divide(m2).asType(Pressure.class);
```

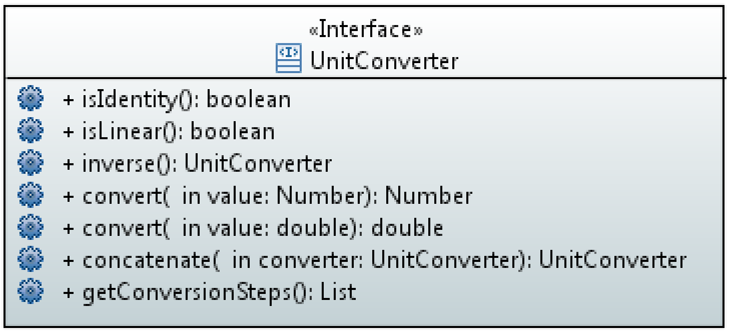
The `asType(Class)` method, which can be applied on a `Unit` or `Quantity` instance, checks at run time if the unit has the dimension of a given quantity, specified as a `Class` object. If the unit doesn't have the correct dimension, then a `ClassCastException` is thrown. This check allows for earlier dimension mismatch detection compared to the unchecked casts, which will throw an exception only when a unit conversion is first requested.

Unit Conversion

Conversion involves two steps: 1) obtain a `UnitConverter` object for a given pair of source and target units, and 2) use it to convert any number of floating point values. The example below converts 4 and 6 metres to 400 and 600 centimetres respectively:

```
Unit<Length> sourceUnit = METRE;
Unit<Length> targetUnit = CENTI(METRE);
UnitConverter converter = sourceUnit.getConverterTo(targetUnit);
double length1 = 4.0;
double length2 = 6.0;
double result1 = converter.convert(length1);
double result2 = converter.convert(length2);
```

This example illustrates the advantages of having a `UnitConverter` interface as opposed to simply including a method in the `Unit` interface along the lines of `convert(double, Unit)`. The process of checking unit compatibility and computing the conversion factor (`Unit.getConverterTo(Unit)`) is a costly operation compared to the conversion itself (`UnitConverter.convert(double)`). This way, the conversion factor needs to be computed only once for a series of floating point values. The `UnitConverter` interface encapsulates the result of this operation. Once available, it can be applied efficiently on a large number of values. Different implementations may exist for different kinds of unit scales (identity, linear, logarithmic, *etc.*).


«Interface»
UnitConverter
+ isIdentity(): boolean
+ isLinear(): boolean
+ inverse(): UnitConverter
+ convert(in value: Number): Number
+ convert(in value: double): double
+ concatenate(in converter: UnitConverter): UnitConverter
+ getConversionSteps(): List

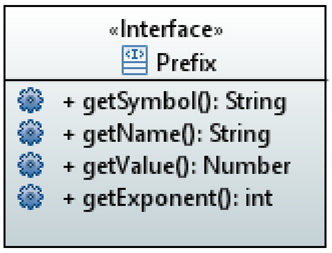
Creating Unit Converters

There is no factory for creating `UnitConverter` instances directly. Converters of different kinds are created indirectly by the various `Unit.getConverterTo(Unit)` methods. For example the two following lines may create the same `UnitConverter`, but the second one is the expected use case since JSR 385 is about unit conversions (as opposed to a general mathematical library):

```
UnitConverter scale1 = new MyCustomConverter(...);
UnitConverter scale2 = KILOMETRE.getConverterTo(METRE);
```

Prefix

A unit prefix is a specifier or mnemonic that is prepended to units of measurement indicating multiples or fractions of the units.



Using the value (base) and exponent of a `Prefix` unit conversion is performed by the `prefix()` method of `Unit`. It is basically a combined call of `getConverterTo(Prefix)` (under the hood) and `transform(UnitConverter)`.

The prefixes of the metric system, like centi, kilo or milli, represent multiplication by powers of ten. In information technology it is common to use binary prefixes, which are based on powers of two.
If the exponent is one, the value alone is used as a multiplication factor (`value`^{1} is the same as `value`) allowing prefixes like quarter multiplying by 0.25 or other non exponential prefixes.

To support the most common use cases in an implementation-agnostic way, both `MetricPrefix` and `BinaryPrefix` are part of the API.

Examples:

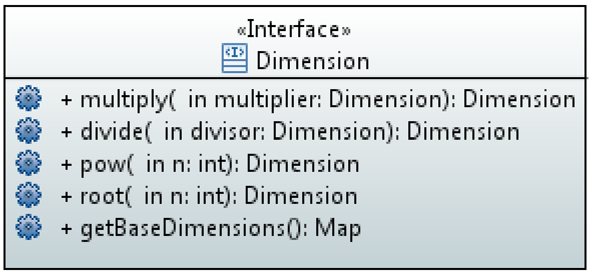
```
Unit<Length> cm = CENTI(METRE);

Is the same as

Unit<Length> cm = METRE.prefix(CENTI);
```

Unit Dimension

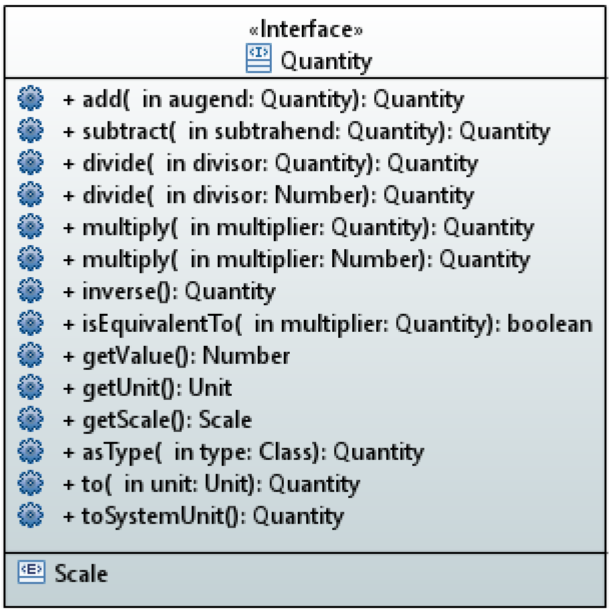
The `Dimension` interface represents the dimension of a unit. Concrete dimensions are obtained through the `getDimension()` method of `Unit`. Therefore, while dimension is a very important aspect of a unit, it is rarely used or instantiated by itself.


«Interface»
Dimension
+ multiply(in multiplier: Dimension): Dimension
+ divide(in divisor: Dimension): Dimension
+ pow(in n: int): Dimension
+ root(in n: int): Dimension
+ getBaseDimensions(): Map

Two units *u1* and *u2* are compatible if they have the same fundamental dimension [DIM_ANALYSIS].

The Quantity Interface

The parent interface for all quantities is `javax.measure.`**`Quantity`**:


«Interface»
Quantity
+ add(in augend: Quantity): Quantity
+ subtract(in subtrahend: Quantity): Quantity
+ divide(in divisor: Quantity): Quantity
+ divide(in divisor: Number): Quantity
+ multiply(in multiplier: Quantity): Quantity
+ multiply(in multiplier: Number): Quantity
+ inverse(): Quantity
+ isEquivalentTo(in multiplier: Quantity): boolean
+ getValue(): Number
+ getUnit(): Unit
+ getScale(): Scale
+ asType(in type: Class): Quantity
+ to(in unit: Unit): Quantity
+ toSystemUnit(): Quantity
Scale

This interface represents a quantitative property or attribute of something. Mass, time, distance, heat and angular separation are among the familiar examples of quantitative properties. This API supports only measurable quantities – all `Quantity` instances are conceptually (`Number`, `Unit<Q>`) tuples.

`Quantity` also supports equivalence via the `isEquivalentTo()` method like the `Unit` interface.

The `getValue()` and `getUnit()` methods return the numeric value of the quantity and the `Unit` the number is represented in. `Quantity` further contains a converter method `to(Unit<Q>)`, converting the value to a desired target unit as well as a conven. The `getScale()` method returns the `Scale` of the quantity.

Quantity Scale

An absolute scale is a measurement that begins at a minimum point, and progresses in only one direction. An **absolute** scale differs from a **relative** scale, which begins at some point selected by a person or group and can progress in both directions. Measurements of `Length`, `Area` and `Volume` are inherently absolute, although measurements of distance are often relative, the height above sea level for example.

Operations on Quantities

Quantities can be created dynamically as the result of algebraic operations on existing quantity objects. Through these operations, JSR 385 provides consistency between the numeric value and corresponding unit of each quantity. Where the `Unit` interface has a matching operation, each of these is called, ensuring its type-consistency checks are applied the same way here. Where appropriate, names of these operations match the `BigDecimal` class in Java SE.

The following table summarizes the algebraic operations provided in the `Quantity` interface.

Result with same dimension	***Result with different dimension***
Binary operations: **`add`** (Quantity) **`subtract`** (Quantity) **`multiply`** (Number) **`divide`** (Number)	*Binary operations:* **`multiply`** (Quantity) **`divide`** (Quantity) **`to`** (Unit)
	Unary operations: **`inverse`**()

The following assignments are examples of compile-time type safe quantity operations:

```
ServiceProvider provider =  ServiceProvider.current();
QuantityFactory<Length> factory = provider.getQuantityFactory(Length.class);
Quantity<Length> m    = factory.create(10, METRE);
Quantity<Length> cm   = factory.create(5, CENTI(METRE));
Quantity<Length> inch = cm.multiply(2.54);
```

For operations with different dimensions the result is a type that cannot be determined statically by the Java type system. The assignments below are not type safe, and require an explicit cast to avoid a compile error. The Java compiler emits an “unchecked cast” warning for such code.

```
QuantityFactory<Mass> massFactory = provider.getQuantityFactory(Mass.class);
Quantity<Mass>    mass    = massFactory.create(1035, KILOGRAM);
Quantity<Volume>  volume  = massFactory.create(1000, LITRE);
Quantity<Density> density = (Quantity<Density>) mass.divide(volume);   // Unsafe!
```

Note: `Density` is not part of the API and only used here as an example. Also see Supported quantities

The compiler warning can be avoided by using the generic <?> wildcard for results, since the dimension and quantity type of the result may be new or unknown:

```
Quantity<?> density = mass.divide(volume);
```

As of Java 8, checks cannot be performed at compile time for such code. However, code can be written in a slightly safer way as follows:

```
Quantity<Density> density = mass.divide(volume).asType(Density.class);
```

The `asType(Class)` method, which can be applied on a `Quantity` instance, checks at run-time, just like it does on a Unit.

Quantity as parameter type

Interfaces extending `Quantity`, sometimes referred to as 'quantity types', are typically used as type parameters to characterize a parameterized class, allowing generics' compile-time type checking to detect mismatches between a parameterized variable and the instantiated value assigned to it.

```
Sensor<Temperature>   sensor        = ... // Generic sensor.
QuantityFactory<Mass> massFactory   = provider.getQuantityFactory(Mass.class);
Quantity<Mass>        mass          = massFactory.create(180, POUND);
Vector3D<Speed>       aircraftSpeed = new Vector3D(12.0, 34.0, -45.5, METRE_PER_SECOND);
```

Supported quantities

`Quantity` subtypes are defined for a representative set of supported units. The table below lists all quantity types defined in `javax.measure.quantity`, together with the name of the `Unit` constant declared in the `Units` class of the reference implementation. The "Base" column states if it's a quantity type of the 7 SI Base Units[7].

This table can be extended by applications or 3rd party libraries with user-defined quantities. However, for any quantity not included in this API, interoperability between the quantities defined in different libraries is reduced. For example, if two independent libraries named "A" and "B" define a `Salinity` quantity in their own respective packages, it is not possible to use a `Salinity` instance of library A in places where a `Salinity` instance of library B is expected. Their associated `Unit<Salinity>` instances cannot be interchanged either, but they are nevertheless compatible if the units use only the standards `Dimension` instances. Consequently quantities of library A are convertible to quantities of library B, and conversely.

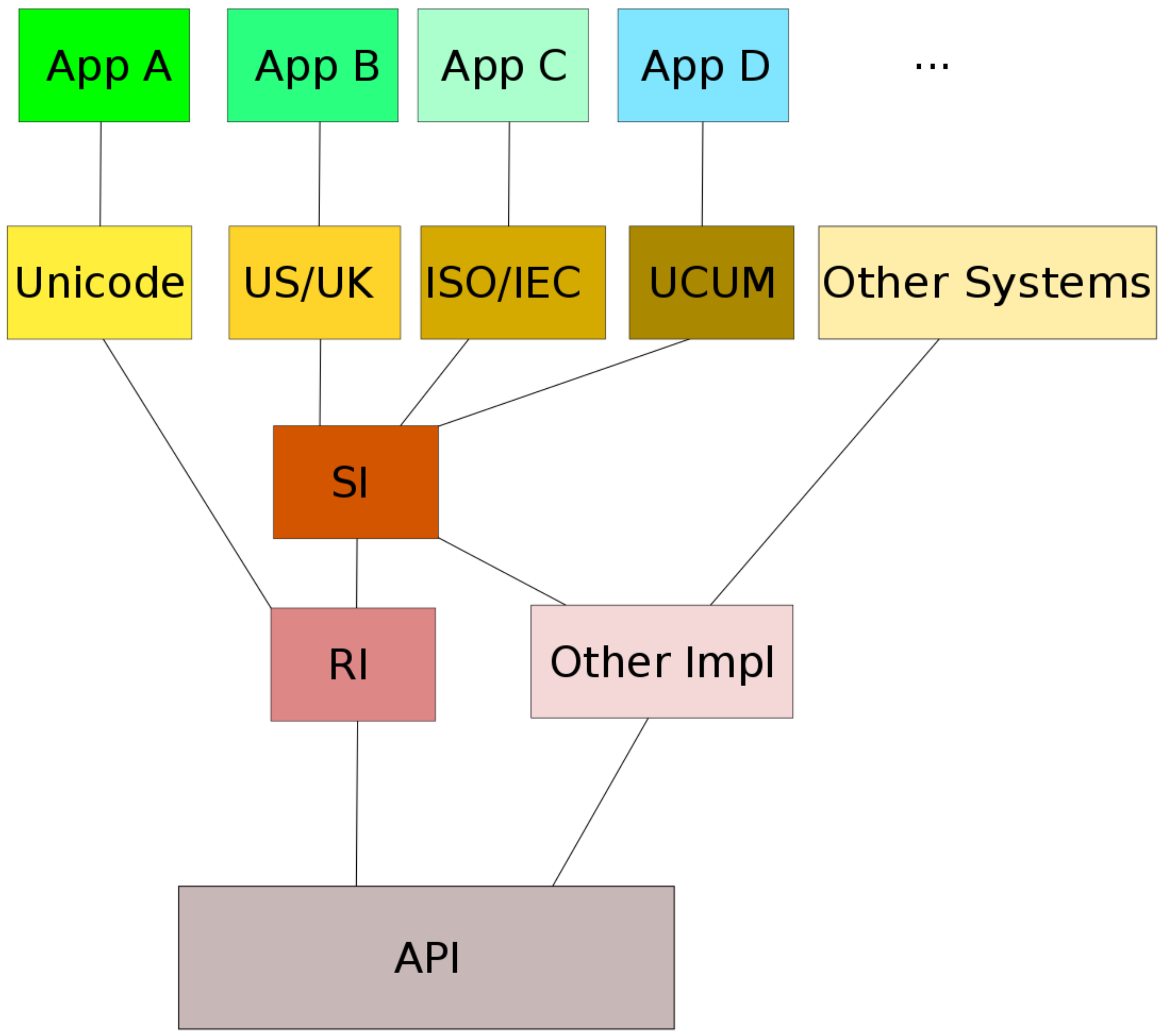
Quantity type	Unit constant	Symbol	Base
`Acceleration`	`METRES_PER_SQUARE_SECOND`	m/s^2	
`AmountOfSubstance`	`MOLE`	mol	Y
`Angle`	`RADIAN`	rad	
`Area`	`SQUARE_METRE`	m^2	
`CatalyticActivity`	`KATAL`	kat	
`Dimensionless`	`ONE`	1	
`ElectricCapacitance`	`FARAD`	F	
`ElectricCharge`	`COULOMB`	C	
`ElectricConductance`	`SIEMENS`	S	
`ElectricCurrent`	`AMPERE`	A	Y
`ElectricInductance`	`HENRY`	H	
`ElectricPermittivity`	`FARAD_PER_METRE`	F/m	
`ElectricPotential`	`VOLT`	V	
`ElectricResistance`	`OHM`	Ω	
`Energy`	`JOULE`	J	
`Force`	`NEWTON`	N	
`Frequency`	`HERTZ`	Hz	
`Illuminance`	`LUX`	lx	
`Length`	`METRE`	m	Y
`LuminousFlux`	`LUMEN`	lm	
`LuminousIntensity`	`CANDELA`	cd	Y
`MagneticFieldStrength`	`AMPERE_PER_METRE`	A/m	
`MagneticFlux`	`WEBER`	Wb	
`MagneticFluxDensity`	`TESLA`	T	
`Mass`	`KILOGRAM`	kg	Y
`Power`	`WATT`	W	
`Pressure`	`PASCAL`	Pa	
`RadiationDoseAbsorbed`	`GRAY`	Gy	
`RadiationDoseEffective`	`SIEVERT`	Sv	
`Radioactivity`	`BECQUEREL`	Bq	
`SolidAngle`	`STERADIAN`	sr	
`Speed`	`METRES_PER_SECOND`	m/s	
`Temperature`	`KELVIN`	K	Y
`Time`	`SECOND`	s	Y
`Volume`	`CUBIC_METRE`	m^3	

The API includes the full list of coherent derived units in the SI with special names and symbols[8] from the [BIPM] SI brochure. As Examples of coherent derived units in the SI expressed in terms of base units[9] in the same brochure states, this table is not a complete list of derived units and quantities. Therefore, we tried to

[7] http://www.bipm.org/en/publications/si-brochure/section2-1.html
[8] http://www.bipm.org/en/publications/si-brochure/table3.html
[9] http://www.bipm.org/en/publications/si-brochure/section2-2-1.html

apply common sense as for which units and quantities to include in the API. Some like `Area`, `Volume`, or `Speed` are widely used, while e.g. refractive index matches a quantity like `Dimensionless` and constant `ONE`, hence we left these out. If explicitly needed, those quantities are subject to extension modules like si-units[10].

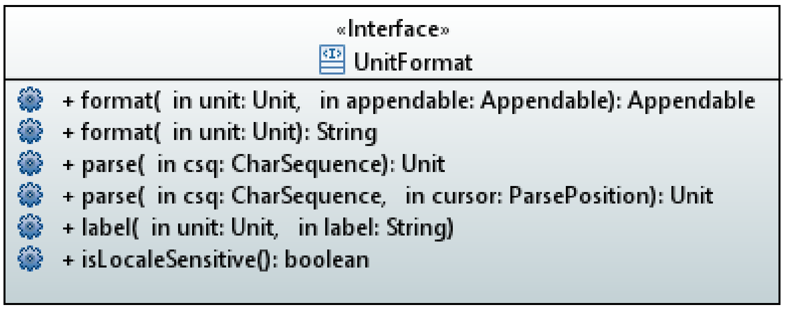
The following diagram[11] shows the modular approach and how applications may use existing modules or create their own on top of the appropriate set of modules:



Unit Formatting

The `UnitFormat` interface allows to `format` a given unit into an `Appendable` and `parse` a specified `CharSequence` to produce a `Unit`. It also offers a simplified `format()` method returning a `String`. If there is no unit to parse, a dimensionless unitary unit is returned.

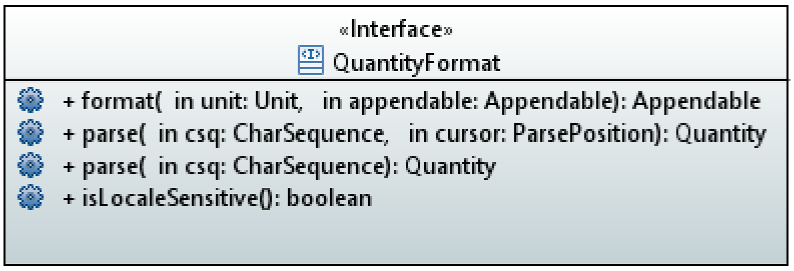
[10] https://github.com/unitsofmeasurement/si-units
[11] extracted from https://github.com/unitsofmeasurement/uom-systems


«Interface»
UnitForm at
+ format(in unit: Unit, in appendable: Appendable): Appendable
+ format(in unit: Unit): String
+ parse(in csq: CharSequence): Unit
+ parse(in csq: CharSequence, in cursor: ParsePosition): Unit
+ label(in unit: Unit, in label: String)
+ isLocaleSensitive(): boolean

In addition, there are 2 methods, `label()` and `isLocalSensitive()`, both assisting format and parse on different environments. `label()` attaches a "system-wide"[12] label to the specified unit. Especially for units that have no symbol or name (e.g. product of units), this offers a standard way to assign a display label to a unit. It also allows to override an existing label (symbol, name or the result of operations) if you need to display that unit differently. `isLocalSensitive()` tells whether this format depends on a locale. In environments that do not support `Locale`, e.g. Java ME, this usually returns `false`. In the reference implementation the `SimpleUnitFormat` class is locale-insensitive while other implementations like `LocalUnitFormat` are locale-sensitive. If a `UnitFormat` implementation `isLocalSensitive()`, it usually provides at least one instance for a given `Locale` or equivalent. There, `label()` will only affect that particular instance and has to be applied for all supported languages or locales.

Quantity Formatting

The `QuantityFormat` interface allows to `format` a given quantity into an `Appendable` and `parse` a specified `CharSequence` to produce a `Quantity`.


«Interface»
QuantityFormat
+ format(in unit: Unit, in appendable: Appendable): Appendable
+ parse(in csq: CharSequence, in cursor: ParsePosition): Quantity
+ parse(in csq: CharSequence): Quantity
+ isLocaleSensitive(): boolean

5. SPI

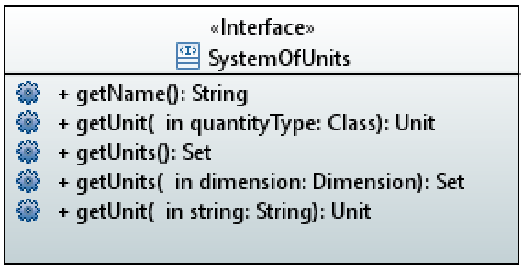
Quantity Factory

`QuantityFactory` applies the [FACTORYPATTERN] to create a new Quantity for the specified number stated in the specified Unit.

System of Units

A system of units is a set of units chosen as standard for specifying measurements. A system contains a small set of well-defined units, called base units, which by convention are regarded as dimensionally independent of other base units. It also contains a larger set of derived units formed as products of exponentiated base units. A widely used unit system is the Metric System, or SI.

[12] applies to the instance of the VM, thread or at least the `UnitFormat` implementation.


«Interface»
SystemOfUnits
+ getName(): String
+ getUnit(in quantityType: Class): Unit
+ getUnits(): Set
+ getUnits(in dimension: Dimension): Set
+ getUnit(in string: String): Unit

`SystemOfUnits` implementations are encouraged (but not required) to expose all their base units and some of their derived units as static final constants. For example the `Units` class in the reference implementation provides static constants like `METRE` (a basic unit) or `WATT` (a derived unit).

Service Provider

`ServiceProvider` is the only class (other than exceptions or enums) in JSR 385. It is an abstract base class for service providers defined by implementations and extension modules on top of them.
`ServiceProvider.current()` provides the currently loaded `ServiceProvider` instance discovered with the highest priority. `ServiceProvider.available()` returns a list of all `ServiceProvider` instances found in the current classloader(s). The default mechanism is based on the `java.util.ServiceLoader` class.
by calling `ServiceProvider.setCurrent()` implementations or 3rd party modules may chose a different mechanism, either OSGi, a Dependency Injection framework or other service loading techniques like Java ME LIBlets [MEEP8_OPTIONAL] as well as Java 9 [JIGSAW].
By ordering the registered instances along the priority (the most significant first), it is also possible to override partial aspects, as the first non null provider is returned by the call.

Services

The following services are available via `ServiceProvider`:

- `FormatService` to retrieve different `UnitFormat` or `QuantityFormat` implementations defined by the implementation in use or 3rd party libraries.
- `SystemOfUnitsService` provides access to different `SystemOfUnits` and `Prefix` implementations defined by either the implementation of choice or 3rd party unit system libraries. Because `Prefix` implementations like `MetricPrefix` are often related to a particular `SystemOfUnits` we considered it a good choice combining access to both in `SystemOfUnitsService`.

Obtaining Unit Instances

`Unit` instances can be obtained in a number of ways.

Units Obtained from Unit Systems

Units can be obtained from different `SystemOfUnits` provided by a `SystemOfUnitsService` implementation. Another easy but less portable way is to directly use one of the predefined constants provided by implementations like the `Units` class in the reference implementation (see Supported quantities). Because such constants are known to the compiler, this approach also provides compile-time checking as discussed in Unit Parameterization. If you know the `String` representation of a `Unit`, you can look it up from a `SystemOfUnits` instance as well.

New units can also be created by applying algebraic operations to existing units (Algebraic Operations), or by parsing a unit symbol (Units Obtained by Symbol Parsing).

Units Obtained by Symbol Parsing

Units can be created dynamically from their string representation – usually their unit symbol, although other kinds of string representations are allowed as well. Bidirectional transformations between units and string representations (parsing and formatting) are performed by implementations of `UnitFormat`. This interface serves three purposes:

- Maintenance of associations between an arbitrary number of unit instances and their string representations. This usually includes all base units together with the derived units that have a special name and symbol. For example, a `UnitFormat` instance for SI symbols should map the "**W**" string to the `WATT` unit instance.
- Mapping of prefix symbols to the corresponding scaling methods, where applicable. For example a `UnitFormat` instance for `SI` symbols should map the "**k**W" string to the `KILO(WATT)` unit.
- Mapping of operator symbols to the corresponding arithmetic methods. For example a `UnitFormat` instance for `SI` symbols should map the "m/s" string to the expression `METRE.divide(SECOND)`.

Users can get `Unit` objects directly from a `UnitFormat` instance. Different formats can be used for recognizing units enumerated in the SI brochure, US customary units or the Unified Code for Units of Measure [UCUM]. 3rd party extensions for unit systems like US customary or UCUM may provide their own `UnitFormat` instance and services or use the default provided by the RI or another JSR 385 implementation.

Examples:

```
FormatService formatService = ServiceProvider.current().getFormatService();
UnitFormat format = formatService.getUnitFormat();
UnitFormat asciiFormat = formatService.getUnitFormat(“Simple”, "ASCII");
UnitFormat ucumFormat  = formatService.getUnitFormat(“UCUM”, "CS");
Unit<Length> metre     = format.parse("m").asType(Length.class);
Unit<Length> foot      = ucumFormat.parse("[ft_i]").asType(Length.class);
Unit<Energy> kilojoule = ucumFormat.parse("kJ").asType(Energy.class);
Unit<Force>  newton    = format.parse("m.kg/s2").asType(Force.class);
Unit<Volume> m3        = format.parse("m³").asType(Volume.class);
Unit<Volume> asciiM3   = asciiFormat.parse("m^3").asType(Volume.class);
```

The `asType(Class)` method checks that the return value of `parse(…)`, which is declared as `Unit<?>`, is of the proper quantity type (e.g. `Unit<Length>`). `Quantity` types were elaborated in Unit Parameterization.

Obtaining Quantity Instances

Quantities Obtained from a Factory

An implementation of `QuantityFactory` can be used to generate simple quantities.

Examples:

```
ServiceProvider  provider =  ServiceProvider.current();
QuantityFactory<Time>   timeFactory = provider.getQuantityFactory(Time.class);
QuantityFactory<Length> lengthFactory = provider.getQuantityFactory(Length.class);
Quantity<Time>    time   = timeFactory.create(12, MILLI(SECOND));
Quantity<Length> length = lengthFactory.create(34.5, MILE);
```

Quantities Obtained from Operations

New quantity instances may also be created as a result of arithmetic operations.

Example:

```
Quantity<Time> timeToDestination(Quantity<Length> distance, Quantity<Speed> speed) {
   return distance.divide(speed).asType(Time.class);
}
```

Quantities Obtained by Symbol Parsing

Quantities can be created dynamically from their string representation – usually number and unit symbol, although other kinds of string representations are allowed as well. Bidirectional transformations between quantities and string representations (parsing and formatting) are performed by implementations of `QuantityFormat`. This interface serves three purposes:

- Maintenance of associations between an arbitrary number of quantity instances and their string representations. For example, a `QuantityFormat` instance using SI symbols should map the string "**1 W**" to a quantity instance of `1 WATT`.
- Mapping of prefix symbols to the corresponding `Prefix` implementations where applicable.
- Mapping of operator symbols to the corresponding arithmetic methods. For example a `QuantityFormat` instance using `SI` symbols should map the "2 m/s" string to an expression like `speedQuantityFactory.create(2, METRE.divide(SECOND))`.

Users can get `Quantity` objects directly from a `QuantityFormat` instance.

Examples:

```
FormatService   formatService =  ServiceProvider.current().getFormatService();
QuantityFormat  format         =  formatService.getQuantityFormat();
QuantityFormat  ndFormat       =  formatService.getQuantityFormat(“NumberDelimiter”);
Quantity<?>     tenMin         =  format.parse("10 min");
Quantity<Mass>  fiveKgTyped    =  format.parse("5 kg").asType(Mass.class);
Quantity<?>     oneMetre       =  ndFormat.parse("1 m");
```

6. Optionality and Dependencies

Embedded Devices (EDs), being a potentially large percentage of platforms for JSR 385, embrace a broad range of device types with very different capabilities. In order to take this into consideration on one hand, and enable a suitable footprint of the implementation on a concrete device (especially those with very limited memory resources) on the other hand, optionality is an important characteristic of JSR 385.

Optionality

Optionality in JSR 385 requires applications to have the possibility to declare which optional components of JSR 385 they do expect in order to be able to run. Oracle outlined reasons and use cases for optionality in [MEEP8_OPTIONAL] Although Java ME Embedded is not adopted and Oracle has made no effort of supporting it unlike Java SE (Embedded) we stick to the same profiles and optionality in JSR 385 as we introduced with JSR 363. Where possible embracing Java 9 JPMS [JIGSAW] modules for that purpose.

Formatting Optionality

The existence of a display, and – if available – its capabilities from a very simple line display without any graphics to a high-end display with graphical capabilities comparable with those of a mobile phone is a degree of freedom for devices using JSR 385.

For a “headless device” (no user interface) there may be neither variable input in need of parsing information, nor equivalent rendering and formatting needs. Therefore, parsing and formatting features provided by the `javax.measure.format` package are optional.

Quantity Types Optionality

The `javax.measure.quantity` package defines quantity types. Compared to the very compact nature of the overall API the number of these quantities makes it the largest package (~20 kb). Specific devices like sensors require only custom quantities, or the memory of these devices would be exceeded by all SI quantity types defined by the `javax.measure.quantity` package. Therefore, the quantity types are optional.

SPI Optionality

The `javax.measure.spi` package defines the Service Provider Interface (SPI) of JSR 385.

JSR 361 (MEEP 8) introduced a LIBlet concept for services on embedded devices, but as an optional feature [MEEP8_OPTIONAL]. Although the minimal version is Java SE 8 for JSR 385, the SPI provided by the `javax.measure.spi` package continues to be optional. Allowing implementations that prefer a completely different approach for storing and retrieving units (e.g. RDBMS, NoSQL databases REST-APIs, etc.) to do this without the need for the SPI.

Dependencies

Aside from the mandatory types defined by the root package `javax.measure` there is only one inter-dependency between optional packages:

- `FormatService` in `javax.measure.spi` refers to `UnitFormat` and `QuantityFormat` in `javax.measure.format`.

Therefore the SPI also requires the format package.

Nothing refers directly to specific types defined by `javax.measure.quantity`, thus it may be used independently of other optional packages on appropriate devices for solutions in need of those quantities.

Profiles

This chapter gives an overview about the possible combination of optional components when implementing JSR 385. Optional components and their dependencies result in the following profiles:

- **Minimal** – Only elements in the “core” (top level) package are used/mandatory.
- **Format** – Core and the optional “format” package on devices and apps in need of formatting/parsing capabilities.
- **Base Quantity** – Elements in “core” plus the 7 SI Base quantities from the “quantity” package.
- **Quantity** – Elements in “core” plus all quantities from the “quantity” package.
- **Quantity & Format** – **Format** and **Quantity** profiles combined (aka everything but **SPI**)
- **SPI** – Elements in “core” and “spi” (which also include “format” as there are SPI services to retrieve `UnitFormat`, see Dependencies)
- **Full** – All API and SPI elements defined by JSR 385.

7. Examples

Unit Conversion

This example shows how units can be converted either using Java base types (`Number` or primitive types) or the conversion method of `Quantity`. Note, elements like `METRE` or `CENTI(METRE)` are based on the RI.

```
ServiceProvider provider = ServiceProvider.current();
QuantityFactory<Length> lengthFactory = provider.getQuantityFactory(Length.class);

Unit<Length> sourceUnit = METRE;
Unit<Length> targetUnit = CENTI(METRE);
UnitConverter converter = sourceUnit.getConverterTo(targetUnit);

double length1 = 4.0;
double length2 = 6.0;
double result1 = converter.convert(length1);
double result2 = converter.convert(length2);

Quantity<Length> quantLength1 = lengthFactory.create(4.0, sourceUnit);
Quantity<Length> quantLength2 = lengthFactory.create(6.0, targetUnit);

Quantity<Length> quantResult = quantLength1.to(targetUnit);
```

Internet of Things

In-Flight Infotainment

This example shows how JSR 385 may be used for an in-flight infotainment system to display the plane's speed, time to destination or other information. In multiple units.

```
ServiceProvider provider = ServiceProvider.current();
QuantityFactory<Length> lengthFactory = provider.getQuantityFactory(Length.class);
Airplane airplane = new Airplane("A380");

Quantity<Length> distance = lengthFactory.create(10427, KILO(METRE));
Quantity<Speed> airplaneSpeed = airplane.getSpeed();

Quantity<Time> timeToDest = distance.divide(airplaneSpeed).asType(Time.class);


System.out.println(airplane + " flying " + airplaneSpeed);
System.out.println(airplane + " flying " + airplaneSpeed.to(MILES_PER_HOUR));
System.out.println("TTD: " + timeToDest.to(HOUR));
```

Quantified Self

Body Mass Index

A small example of how JSR 385 can be used for the Quantified Self to calculate a person's Body Mass Index (BMI):

```
ServiceProvider provider = ServiceProvider.current();
QuantityFactory<Length> lengthFactory = provider.getQuantityFactory(Length.class);
QuantityFactory<Mass> massFactory = provider.getQuantityFactory(Mass.class);

Quantity<Length> height = lengthFactory.create(1.83, METRE);
```

```
// for US measure units, the above line can be replaced by the following:
// Quantity<Length> heightUS = lengthFactory.create(6, FOOT);
// Quantity<Length> height = heightUS.to(METRE);

Quantity<Mass> mass = massFactory.create(85, KILOGRAM);
// for US measure units, same could be done from lb as above

Quantity<Area> squareHeight = height.multiply(height).asType(Area.class);
Quantity<?> bmi = mass.divide(squareHeight);
```

DevOps & Cloud

Monitoring

Performance Co-Pilot [PCP] is a very popular performance monitoring and analysis framework. Started by Silicon Graphics in 2000, then continued by Aconex it is now maintained by Red Hat.

Leading players and Cloud innovators like Netflix build upon it with its [VECTOR] performance visualization tool

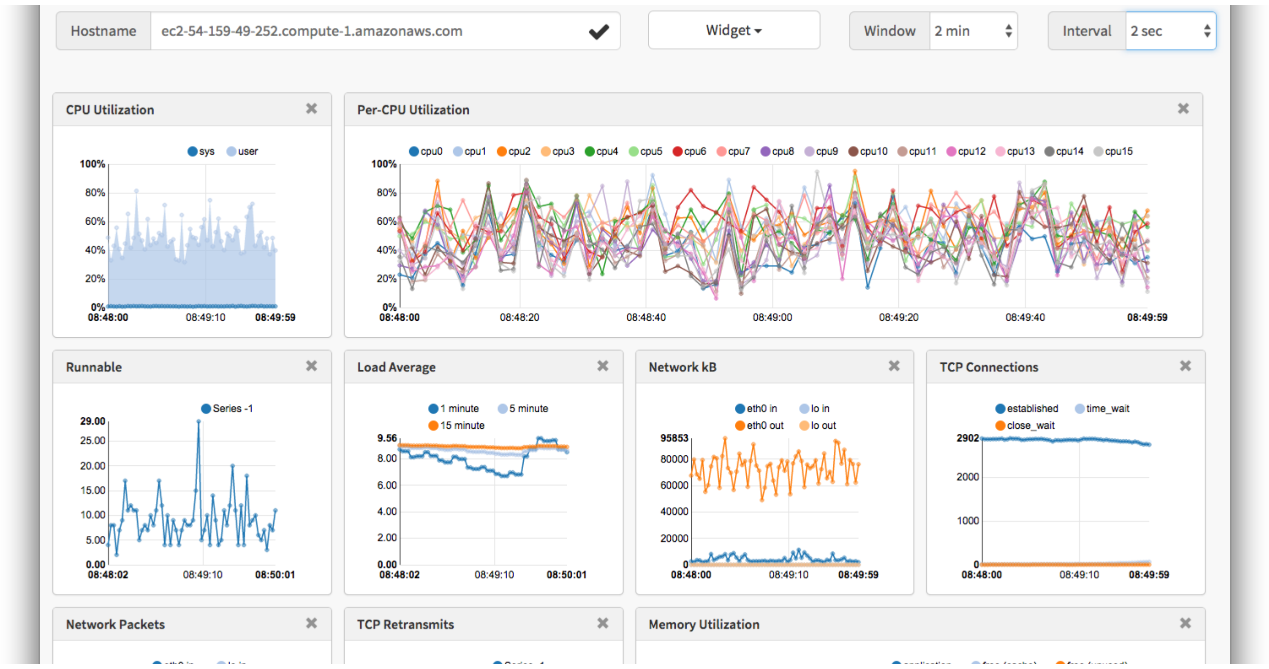


Image © 2015 Netflix

Units are very important to Performance Co-Pilot. Each metric has an associated descriptor providing information that may be used to decode and interpret values of the metric over time. The descriptor provides the following information:

- A unique internal Performance Metric Identifier (PMID)
- The data type for the value(s), being one of 32, U32, 64, U64, FLOAT, DOUBLE, STRING, AGGREGATE.
- The identifier for the associated instance domain for set-valued metrics, else PM_INDOM_NULL for singular metrics.
- The semantics of the value(s), i.e. counter, instantaneous, discrete.
- The units of the value(s), expressed as a dimension and scale in the axes time, space and events.

An example:

```
$ pminfo -md filesys.free
filesys.free PMID: 60.5.3
  Data Type: 64-bit unsigned int  InDom: 60.5 0xf000005
  Semantics: instant  Units: Kbyte
```

Parfait

Parfait [PARFAIT] is a performance monitoring library for Java which extracts metrics and makes them available in a variety of ways (including JDBC, Spring, JMX and the open-source cross-platform [PCP]).

Parfait used [JSR-275] since its inception and following the Public Review approval changed to JSR 363 with version 0.4. Starting with version 1.1 Parfait updated to JSR 385.
The `Monitorable` interface refers to The Unit Interface defined by this JSR.

```
/**
 * @return the JSR-385 Unit represented by the value of this Monitorable.
 *         This may be used to do comparisons and rate-conversions between
 *         metrics which do not share the same scale. Values which do not
 *         take a unit should use {@link tech.units.indriya.AbstractUnit#ONE};
 *         values for which no unit is sensible (e.g. String values) may
 *         return null.
 */
Unit<?> getUnit();
```

Micrometer

Micrometer [MICROMETER_AM] is a performance monitoring library by VMWare used in Spring/Spring Boot and several other frameworks.

While optional it allows to add base units to each metric for maximum portability .

```
Counter counter = Counter
    .builder("counter")
    .baseUnit("beans") // optional
    .description("a description of what this counter does") //
optional
    .tags("region", "test") // optional
    .register(registry);
```

Elasticsearch

Elasticsearch [ELASTIC_TIME] currently supports the following quantities:

- Time
- Data (Information Rate)
- Distance (Length)

Let's look for the nearest big cities near El Cerrito, CA, a city neighboring Berkeley in the San Francisco Bay Area. The lat/long of El Cerrito is 37.9174, -122.3050.

```
curl -XGET 'http://localhost:9200/us_large_cities/city/_search?pretty=true' -d '
{
  "query": {
    "filtered" : {
        "query" : {
            "match_all" : {}
        },
        "filter" : {
            "geo_distance" : {
                "distance" : "20km",
                "location" : {
                    "lat" : 37.9174,
                    "lon" : -122.3050
                }
            }
        }
    }
  }
}'
```

You may notice under [ELASTIC_TIME] and below that unlike [UCUM] Elasticsearch permits a certain ambiguity in unit definitions. Both. `"distance" : "20m"` and `"duration" : "20m"` are legitimate. The first quantity would be `20 metres` the other `20 minutes`. In the context of an entire query it usually makes sense, so it should be possible to tell them apart for both code and human readers.

Configuration

YAML

[YAML], a human friendly data serialization standard for all programming languages, is used for configuration of many systems. Taking the example of [SOG], a vertical, one-dimensional, coupled physics-biogeochemical model for deep estuaries, developed by the Department of Earth, Ocean and Atmospheric Sciences at the University of British Columbia, we'll naturally find plenty of units to configure this application.
[SOG_YAML] shows how YAML is used to write SOG infiles. For example:

```
# SOG code infile template
# SOG code infile for 356.5 day run starting at cruise 04-14 station
# S3 CTD cast (2004-10-19 12:22 LST).

grid:
  model_depth:
    value: 40
    units: m
    variable_name: grid%D
```

HOCON

HOCON (Human-Optimized Config Object Notation) similar to Elasticsearch offers a limited number of units in Typesafe Config, a Configuration library for JVM languages by Lightbend, see [HOCON_UNITS].

Allowing to use duration and certain other units (mostly information or information rate) for cases like:

```
standard-timeout = 10ms
longer-timeout = 30ms
foo.timeout = ${standard-timeout}
bar.timeout = ${longer-timeout}
```

While bar isn't an SI unit (see [BIPM]), using JSR 385 a configuration DSL could also look like:

```
standard-pressure = 10bar
higher-pressure = 30bar
foo.pressure = ${standard-pressure}
bar.pressure = ${higher-pressure}
```

Following the "Foo Bar" pattern of the HOCON example. Of course other Supported quantities may be used out of the box, too, or you can define your own quantities.

Archaius and Apache

Netflix (again) Archaius [NETFLIX_ARCH] extending Apache Commons Configuration [COMMONS_CONF] comes with a dynamic type system. This example from [NETFLIX_IEP] shows how [JSR-310] precursor JodaTime can be used to return `DateTime` or `Duration` values.

```
import org.joda.time.DateTime;
import org.joda.time.Duration;

public class ConfigurationTests {
  interface TestConfig extends IConfiguration {
    @DefaultValue("string")
    public String getString();

    [...]

    @DefaultValue("2014-08-01T00:00:00")
    public DateTime getDateTime();

    @DefaultValue("PT5M")
    public Duration getDuration();
  }
```

Leveraging JSR 385 it'll look like this returning `Quantity` and `Unit`:

```
import javax.measure.Quantity;
import javax.measure.Unit;

public class ConfigurationTests {
  interface TestConfig2 extends IConfiguration {
    @DefaultValue("string")
    public String getString();

    [...]

    @DefaultValue("10 ms")
    public Quantity getQuantity();

    @DefaultValue("bar")
    public Unit getUnit();
  }
```

8. Related Work

The library presented in this API takes inspiration by some sources:

Scientific and Engineering C++ [BART_NACKMAN] presents the implementation strategy used in this framework, which is to derive every kind of unit from a small set of base units raised to arbitrary integer power. Barton and Nackman provide type safety by using the C++ `template` feature in a way that cannot be transposed to Java, nevertheless their principles still apply.

University Corporation for Atmospheric Research (UCAR)[13] implements a Java units library based on `Unit` and `Converter` types, which are the foundation of this API (despite `Quantity` being the facade that many users will see). The UCAR library is used in the NetCDF Java products, which are extensively used in the oceanographic and meteorological communities.

Brochure on the International System of Units defines base and derived units, and provides the minimal set of units that this API expects to be present in every implementation.

Curl, a dynamic language for Web Applications and RIA development which includes Units of Measure support, especially `Quantity` and `Unit` as the two main parts of the API [CURL_QUANT] and CURL_WHIRL]. While the language is not in all parts comparable to Java, Unit support is to a large extent.

Andrew Kennedy, both with his thesis and works on the topic [KENNEDY1996] as well as more recent implementations especially using functional languages like F# [AKENN].

Walter E. Brown presented an SI Library of Unit-Based Computation for C++ in 1998 [FERMILAB98/328] something C++ libraries like [BOOST] or the introduction of user-defined literals [CPP_N2765] made much easier since.

Martin Fowler analyzed this problem [FOWLER1996] in the days before Java 5 parameterized types were available. Together with Igor Janicijevich he further analyzed these patterns in [FOWLER1999].

Works on other Object-Oriented languages like Smalltalk [ACONCAGUA] or [INGALLS]. Some language elements like multiple polymorphism made this work out of JSR 385 scope.

Many contributions came from JSR 275 [JSR-275] and JSR 108 [JSR-108], earlier JCP approaches to Units of Measurement in Java.

While specific to Date and Time only and not compatible with Java ME, [JSR-310] for a single quantity shows a few similarities and terms that look like they're inspired by unit-related JSRs like `TemporalUnit` (see Java 8)

Java language changes – like the ones proposed in *Object-Oriented Units of Measurement* [ALLEN2004] and *A Java Extension With Support for Dimensions* [DELFT1999] – are out of scope for this API.

Other API

JDK

Java 5 introduced a `java.util.concurrent.`**`TimeUnit`** enum which overlaps the purpose of the `Unit<Time>` construct defined in this API. The two constructs could be made interoperable by the addition of the following methods to a Unit implementation:

- `public static Unit<Time> from(TimeUnit timeUnit)`
- `public TimeUnit toTimeUnit()`

Every `TimeUnit` constant can be represented as a `Unit<Time>` instance, but not every `Unit<Time>` instance has an equivalent `TimeUnit`.

[13] http://www.unidata.ucar.edu/software/netcdf-java/v4.1/javadocAll/ucar/units/package-summary.html

Alternatively `TimeUnit` could implement `Unit<Time>`, but this approach would have the following inconveniences:

- Addition of many non-trivial methods in `TimeUnit`.
- Conversions between two `Unit<Time>` are slower than conversions between two `TimeUnit`, because `Unit<?>` is a more generic class which needs to check the dimension of the source and target units. In contrast, TimeUnit doesn't need to perform such checks as it is restricted to the time dimension.

`Unit<Time>` and `TimeUnit` serve different purposes. The `TimeUnit` enum is used in contexts where performance is critical, since it is an argument given to methods controlling execution time with nanoseconds precision. Using an enum, restricted to the time dimension and a universe of six `TimeUnit`, is efficient. On the other hand, `javax.measure.Unit` is a generic and extensible framework used to measure data even in a real time where required.

Time units are particular since they are the only units in the dimension of the quantity that performance concerns try to minimize (neglecting the memory dimension on the assumption that `Unit` instances are small enough). `TimeUnit` may deserve a special status because of that.

Java 8

As mentioned [JSR-310] the JCP Date & Time API for Java SE 8 shows a few similarities we want to list without going into specific detail about them. The RI of JSR 385 explores the synergies with JSR 310 where appropriate. Somewhere along the lines of:

- `public static Unit<Time> from(TemporalUnit timeUnit)`
- `public static Quantity<Time> getQuantity(LocalTime localTimeA, LocalTime localTimeB)`
- `of(Number, TemporalUnit)`

similar to `Instant` handling by `java.util.Date` in Java 8.

Beside the basic idea of modeling the Time/Duration quantity with relevant units being a subset of mostly SI or ISO units similar to those used in this API, there are 3 main points of interest:

- **`TemporalUnit`**
  Not only called a unit but also very similar in aim and name to the enum Java Concurrency API, or in a very minimalistic way the `Unit` in JSR 385.

- **`TemporalField`**
  This, while trying to sound and feel a bit like the old `java.util.Calendar` field attributes, actually has many elements of a `Quantity`:

```
 /**
  * The amount of the period.
  */
private final long amount;
    /**
     * The unit the period is measured in.
     */
    private final TemporalUnit unit;
```

  with equivalent getter and setter methods where `getUnit()` is even identically named.

- **`TemporalAmount`**
  Interconnected with `TemporalUnit`, this is not related to `TemporalField` in JSR-310. Possibly a slightly different interpretation, or work in progress? The README for the JSR says `Duration` or `Instant` are mostly for machine purposes while other types in its extensive `calendar` package are meant for human interaction like UI or as the name says calendars.

- **`Chronology`**
  With the fewest methods and its main sub-class declared “The ISO-8601 calendar system”, this is a System of (Period/Time) Units. Its concrete subclasses all act and behave very much like Systems of Units.

Like seen in other areas and a main idea of standardization like that by the JCP interoperability or at least something like a “bridge” would benefit both APIs. We provide such a bridge in the Reference Implementation of JSR 385.

Java 16

Java 16 brought the `record` type [JEP395] into production, which was first introduced as a preview in Java 14. Records are a natural choice for value oriented data types. For example a `Person` record:

```
public record Person(String firstName, String lastName, Quantity<Length> height,
Quantity<Mass> mass) {
      public String getFullName(){
            return String.format("%s %s", firstName, lastName);
      }
}
```

We can define a `QuantityRange` as a record:

```
public record QuantityRange<Q extends Quantity<Q>>(Quantity<Q> minimum, Quantity<Q>
maximum) {
      public boolean contains(final Quantity<Q> q) {
            Objects.requireNonNull(q);
            return q.getValue() != null && q.getUnit() != null &&
                  q.getUnit().equals(minimum.getUnit()) &&
                  q.getUnit().equals(maximum.getUnit()) &&
                  minimum.getValue().doubleValue() < q.getValue().doubleValue() &&
                  maximum.getValue().doubleValue() > q.getValue().doubleValue();
      }
}
```

Note: this record uses a simplified comparison via `doubleValue()`, in actual implementations including future Multi-Release JAR versions of the JSR 385 RI, we use more robust comparisons and value checks on any `Number`.

And check if the person is within a certain range:

```
var person1 = new Person("John", "Doe", getQuantity(1.8, METRE),
  getQuantity(80, KILOGRAM));
System.out.println(person1);
var min = getQuantity(50, KILOGRAM);
var max = getQuantity(100, KILOGRAM);
var range = new QuantityRange<Mass>(min, max);
if (range.contains(person1.mass())) {
      System.out.println("mass within range.");
} else {
      System.out.println("mass not within range.");
}
```

Although the quantity model is not as flexible and extensible as the JSR 385 RI, it is even possible to implement `Quantity` using a record:

```
public record TemperatureQuantity(double value, TemperatureUnit unit) implements
Quantity<Temperature> {
      @Override
      public Quantity<Temperature> add(Quantity<Temperature> addend) {
          final TemperatureQuantity dn = new TemperatureQuantity(Double.valueOf(
            this.value + addend.getValue().doubleValue()), this.unit);
          return dn;
      }

      @Override
      public Quantity<Temperature> divide(Number divisor) {
            return new TemperatureQuantity(value / divisor.doubleValue(), unit);
```

```
        }

        public Quantity<?> divide(Quantity<?> that) {
                return divide(that.getValue());
        }

        public Quantity<Temperature> multiply(Number that) {
                return new TemperatureQuantity(value * that.doubleValue(), unit);
        }

        public Quantity<?> multiply(Quantity<?> that) {
            return new TemperatureQuantity(value * that.getValue().doubleValue(), unit);
        }

        public Quantity<Temperature> subtract(Quantity<Temperature> that) {
            final TemperatureQuantity dn = new TemperatureQuantity(
             this.value- that.getValue().doubleValue(), this.unit);
            return dn;
        }

        public Number getValue() {
            return value;
        }

        public Unit<Temperature> getUnit() {
            return unit;
        }

        public Quantity<Temperature> inverse() {
            return convert((TemperatureUnit)unit.inverse());
        }

        public Quantity<Temperature> to(Unit<Temperature> unit) {
                if (this.unit.equals(unit)) {
                return this;
                }
                if (unit instanceof TemperatureUnit) {
                        return convert((TemperatureUnit)unit);
                } else {
                        throw new ArithmeticException("Cannot convert " + this.unit + " to "
                + unit);
                }
        }

        public Quantity<Temperature> negate() {
                return new TemperatureQuantity(-value, unit);
        }

        private TemperatureQuantity convert(TemperatureUnit newUnit) {
            return new TemperatureQuantity(value /
                newUnit.getFactor(), newUnit);
        }

        private double doubleValue(Unit<Temperature> unit) {
            final Unit<Temperature> myUnit = getUnit();
            try {
                UnitConverter converter = unit.getConverterTo(myUnit);
                return converter.convert(getValue().doubleValue());
            } catch (UnconvertibleException e) {
                throw e;
            }
        }

        [...]
}
```

Note: this record also uses a simplified `double` value representation. In some cases measurements are perfectly fine with double / floating point values, in others a higher precision can be required.

Other JSRs

JSR 256

The JSR 256 Mobile Sensor API [JSR-256] allowed J2ME application developers to fetch data from sensors. A sensor is any measurement data source. Sensors can vary from physical sensors such as magnetometers and accelerometers to virtual sensors that combine and manipulate the data they have received from various kinds of physical sensors. An example of a virtual sensor might be a level sensor indicating the remaining charge in a battery or a field intensity sensor that measures the reception level of the mobile network signal in a mobile phone. While containing the notion of Unit in its API, JSR 256 lacked not only true support for dimension, quantity or other important aspects (which were at most mentioned by the spec document, but not implemented) but also unit type-safety even at runtime. Completely unrelated units could be mixed with numeric data. The spec happens to refer to [JSR-108] btw., so at least for the Unit interface one can say it must have been vaguely inspired by 108.

After a complete redesign of event handling, power management and other aspects in Java ME 8 as well as a correlated Device I/O project (under OpenJDK) JSR 256 has become outdated. Als former Spec Lead Nokia now part of Microsoft Mobile keeps this JSR others merely as J2ME legacy still used by some low end devices like the Asha Series. There is no further development to be expected on any of these and if Microsoft was to offer future phones using Java ME 8 or above they would likely not support JSR 256.

JSR 308

JSR 308 came with Java SE 8, but none of its annotations were so far used by the JDK or other JSRs. It remains up to particular applications to use it. Since it will not work with Java ME Embedded and is not supported by any part of the Java platform other than the compiler so far, we will not take it into consideration. There’s the so-called Checker Framework, an open source PoC for JSR 308 which is not part of the actual JSR. It also contains a rudimentary unit checker, see [JSR-308_UNITS].

While operations like:

```
@m int meters = 5 * UnitsTools.m;
@s int secs = 2 * UnitsTools.s;
@mPERs int speed = meters / secs;
```

could potentially help fill the small gap with divide() or multiply() operations on *Unit* or *Quantity* if the argument is another unit or quantity, the unit checker so far seems restricted to Java primitive types and operations like + - / or *, etc. not complex types like Double or Quantity, at least there is neither evidence nor examples by the JSR 308 team that it might work. It is extremely cumbersome to apply these checkers in your own solution, especially if you wanted to extend beyond the simple SI base type selection the unit checker seems to provide at most.

```
 javac -processor org.checkerframework.checker.units.UnitsChecker \
       -Aunits=myproject.qual.MyUnit,myproject.qual.MyOtherUnit MyFile.java ...
```

means one must list every single custom unit in the command line of the compiler, so far there is no IDE support or other help e.g. via Maven, Gradle, etc. given JSR 308 remains a rather exotic niche at the moment, even those working on the JDK itself. It may take until Java 20 or above to get eagerly awaited features like value types where type annotations that now work on a primitive type should work just the same way on value types. The API of JSR 385 is based almost entirely on interfaces which according to [JEP169] will work fine for value types to implement once Java supports these new types.

Most importantly JSR 308 and all checkers are limited to **Compile Time**, meaning just like F# (see [AKENN]) any knowledge of a unit or quantity is lost at runtime. Which the vast majority of JSR 385 (except generic type arguments like <Q> that Java so far doesn't preserve at runtime in lack of "reification", another eagerly awaited feature for future Java versions;-) won't lose. While offering similar compile-time unit type-safety for most operations except a few operations like dividing by another unit/quantity where the result is not always known, except for well-defined cases (SI types) which allow the asType() operator as a runtime harness.

As of now until JSR 308 and standard annotations were at least offered as an optional module in a post [JIGSAW] Java 9 or 10 it can only be seen as an additional option for some very distinct use cases and applications, only on Java SE or EE 8 and above. Thus we will not take it into consideration for JSR 385 RI, also see Source Code and Binary Compatibility. Other implementations may use it where feasible.

JSR 354

Calculate banner clicks in a certain period or how many gold medals their team won during the Olympic Games. All those are quantities and while it exceeds the scope of the spec, not only [AKENN] or [ACONCAGUA] clearly define monetary units and the financial industry as one of their target groups.

As a result, the Money and Currency API for the Java Platform [JSR-354] was accepted by the Java Community Process in 2012, passed Public Draft 2 in May 2014 and went Final a year later in May 2015.

Financial libraries may leverage synergies between JSR 385 and 354, e.g. for calculations like "Cost per gallon fuel", etc. but the somewhat different and more complex nature of currency conversion compared to unit conversions between well defined unit systems means, this is out of scope for this JSR or its RI.

ICU4J

This open library [ICU4J] driven and developed by an industry group of companies like IBM, Oracle, Adobe, Apple or Google has created a library for Java that is used in many parts of Eclipse projects and also other platforms like Android.

Where Measure and Unit types come into place they are visible (despite OSGi at least in Java) but neither documented much nor recommended to be used. The only purpose at the moment is being used by enhanced and improved copies of `java.util.Currency` or other types, especially those relevant to localization. Therefore minor parts (only the Measure and MeasureUnit) seem to overlap with JSR 385. Especially the Quantity Interface not being a concrete class may allow usage by the ICU4J Measure class. While not a reason for removing types like Measure from our prior approach it indirectly makes usage more appealing without having to rename or remove parts of its API. The fact, MeasureUnit is not named Unit like in JSR 385 would potentially allow its adoption by future ICU versions. Minor differences in the Quantity/Measure exist like its value method called getNumber(), again other approaches like [JSR-310] simply call it get(), but you must choose from an array of units first.

OSGi Measurement

OSGi Measurement [OSGIM], since OSGi 5 part of an add-on Bundle called "Compendium" consists of just 3 classes:

- Measurement: Represents a value with an error, a unit and a time-stamp.
- State: Groups a state name, value and timestamp.
- Unit: A unit system for measurements.

As their descriptions, especially the latter already tell, *Unit* mixes the actual type with the most common constants in the base SI system (very similar to [ICU4J] or [JSR-256] both created somewhat around a similar time btw.) Unlike these two other libraries OSGi Measurement had neither means nor intention to be extended beyond these. *Measurement* and *State* somewhat overlap, but the state is meant to be some sort of enum with a name and value (much like Java enums later) plus a timestamp while Measurement groups the *Quantity* concept of JSR 385 or [FOWLER1996] and [FOWLER1999] together with a timestamp into a single class. OSGi Measurement has not been actively maintained for many years. It provides runtime checking of the most basic SI types while everything else that exceeds its scope normally results in a `RuntimeException`.

9. Frequently asked questions

Why are units parameterized with Quantity? Wouldn't Dimension be more appropriate?

The first concept from which all others are derived is Quantity. Quantities are suitable to parameterize units because they answer an important question at compile time: What kind of quantity a unit represents. The standard parameterization mechanism in Java works reasonably well with quantities for most operations. For example, a length unit after being scaled is still a length unit. Using a Dimension class for parameterization of units can lead to problems:

Dimensions change with the model. For example the dimension of the Watt unit is $[L]^2 \cdot [M]/[T]^3$ in the standard model (SI system of units), but become [M]/[T] in the relativistic model.

Units may have the same dimension and still apply to different quantities. For example both Torque and Energy have a dimension of $[L]^2 \cdot [M]/[T]^2$ in the standard model. Nevertheless it is convenient and safer to consider them as two different quantities with their own units. Other examples are sea water salinity (PSS-78), some kind of concentration and angles, which are all dimensionless but still convenient to treat as different kinds of quantities.

Users will work primarily with `Quantity` and `Unit` objects. The need to work with `Dimension` objects is less common. If the units were parameterized with dimension instead of quantity, the API would need to define many `Dimension` subtypes in the same way that it currently defines many Quantity sub-interfaces, resulting in a large increase of API size – almost doubling the amount of types.

Why doesn't the API use more Annotation types?

[EFFECTIVEJAVA] highlights that defining new types is better done by interfaces than annotations. While using annotation-based JSRs like CDI or Bean Validation together with JSR 385 seems a proper choice. Allowing full usage of new custom types and unit-safety. This said, annotations in such areas will have their place and purpose, but the API as such defines the types for this first. Unfortunately Java Annotations currently won't accept most Java types other than `String`, enums or primitives. When we helped with the design and development of the MicroProfile Metrics [MP_METRICS] feature, it became obvious that a custom type like `Time`, `Length`, etc. may not be used inside an annotation like `@Metered` or `@Counted` of MP Metrics. Instead its `unit()` attribute holds the string representation of involved units. This limits the direct use of the API inside annotations quite a bit. Unless the Java language may change some day, the only way is to parse or format strings where unit information is required inside an annotation.

Why doesn't the API provide a Measurement interface?

Measurement is sometimes defined as the set of operations executed for determining the value of a quantity. Some applications record the instant when those operations were done, as a timestamp associated with the measurement. Many applications associate precision or quality information to measurement, but the ways to encode that information vary greatly. For example the precision may be represented by a 95% confidence interval or merely an alphabetic code. Such diversity of information makes it difficult to define a universal set of properties associated with measurement.

Another difficulty is to choose a proper location in the type hierarchy. Measurement is sometimes considered as a quantity defined by a measurement process, which would tend toward defining `Measurement` as a subtype of `Quantity`. But JSR 385 defines `Quantity` as a scalar value (a `Number`) with a `Unit`, leaving other kinds of values out of scope, such as vector data or quantities describing a "multitude" (like "few, some, many" or "S, M, XL" commonly applied by clothing measurements among others) Java already offers an `enum` for that, so we see no need to address these types of quantities. Such restrictions would apply to wind or current measurements for instance. Using records, it is also very easy to define a `MeasurementRecord` like this:

```
public record MeasurementRecord<Q extends Quantity<Q>>(Quantity<Q> quantity,
 Instant instant) {
}
```

References

[ACONCAGUA]	Arithmetic with Measurements on Dynamically-Typed Object-Oriented Languages, Hernán Wilkinson, Máximo Prieto, Luciano Romeo, OOPSLA'05, October 16–20, 2005.
[AKENN]	Andrew Kennedy: Units of Measure
[ALLEN2004]	Allen E., Chase D., Luchangco V., Maessen J.-W. *and* Steele G.L. Jr., 2004. *Object-Oriented Units of Measurement.* Sun Microsystems Laboratories.
[BART_NACKMAN]	Barton J.J *and* Nackman L.R., 1994. *Scientific and engineering C++ – an introduction with advanced techniques and examples.* Addison-Wesley.
[BIPM]	*Bureau International des Poids et Mesures* - Brochure on the International System of Units: http://www.bipm.org/en/publications/si-brochure/
[BOOST]	Matthias C. Schabel, Steven Watanabe, Chapter 32. Boost.Units 1.1.0 http://www.boost.org/doc/libs/1_51_0/doc/html/boost_units.html
[CGPM]	Meetings of the CGPM: https://www.bipm.org/en/committees/cg/cgpm/meetings
[CLDR]	CLDR - Unicode Common Locale Data Repository: http://cldr.unicode.org/
[COMMONS_CONF]	Apache Commons Configuration: https://commons.apache.org/proper/commons-configuration/index.html
[CPP_N2765]	C++ proposal N2765 (user-defined literals) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2765.pdf
[CURL_QUANT]	Curl – Quantities and Units
[CURL_WHIRL]	Taking Curl for a Whirl
[DELFT1999]	Delft, A., 1999 – A Java Extension With Support for Dimensions. Software - Practice and Experience 29(7).
[DIM_ANALYSIS]	Wikipedia – Dimensional analysis: https://en.wikipedia.org/wiki/Dimensional_analysis
[ELASTIC_TIME]	Elasticsearch – Common Options Time Units: https://www.elastic.co/guide/en/elasticsearch/reference/current/common-options.html#time-units
[EFFECTIVEJAVA]	Bloch, Josh, Effective Java Second Edition, http://www.oracle.com/technetwork/java/effectivejava-136174.html
[EQUIVALENT]	Cambridge Dictionary – equivalent https://dictionary.cambridge.org/dictionary/english/equivalent
[FACTORYPATTERN]	Wikipedia – Factory method pattern: https://en.wikipedia.org/wiki/Factory_method_pattern
[FERMILAB98/328]	Walter E. Brown, 1998 – Introduction to the SI Library of Unit-Based Computation http://lss.fnal.gov/archive/1998/conf/Conf-98-328.pdf
[FOWLER1996]	Fowler, M. Analysis Patterns: Reusable Object Models, Addison-Wesley, Reading, MA, 1996.
[FOWLER1999]	Igor Janicijevic and Martin Fowler: Analysis Patterns, http://martinfowler.com/apsupp/apchap3.pdf, http://martinfowler.com/apsupp/apchap4.pdf, http://martinfowler.com/apsupp/appfacades.pdf

[HOCON_UNITS] HOCON Units format:
https://github.com/typesafehub/config/blob/master/HOCON.md#units-format

[ICU4J] ICU4J Project http://icu-project.org

[INGALLS] Ingalls, D. A simple technique for handling multiple polymorphism. ACM, SIGPLAN Notices, 21(11):347—349, Nov. 1986.

[ISO] International Organization for Standardization
ISO 31 (Quantities and units)

[JEP169] JEP 169 – Value Objects: http://openjdk.java.net/jeps/169

[JEP238] JEP 238 – Multi-Release JAR Files: https://openjdk.org/jeps/238

[JEP395] JEP 395 – Records: https://openjdk.org/jeps/395

[JIGSAW] OpenJDK – Project Jigsaw: http://openjdk.java.net/projects/jigsaw/

ISO 1000
ISO 10303 STEP Part 41 . STEP Part 41 EXPRESS Schema
ISO/IEC_80000

[JSR-108] JCP proposal JSR 108
Source Code at http://jsr-108.sourceforge.net/javadoc/javax/units/Unit.html
JSR details: http://jcp.org/en/jsr/detail?id=108

[JSR-256] Mobile Sensor JSR: http://jcp.org/en/jsr/detail?id=256

[JSR-275] JCP proposal JSR 275
Source Code at https://github.com/unitsofmeasurement/jsr-275
JSR details http://jcp.org/en/jsr/detail?id=275

[JSR-308_UNITS] Checker Framework for JSR 308: Units Checker,
http://types.cs.washington.edu/checker-framework/current/checker-framework-manual.html#units-checker

[JSR-310] Date & Time JSR: http://jcp.org/en/jsr/detail?id=310

[JSR-354] JSR 354, Source Code at http://javamoney.org
JSR details http://jcp.org/en/jsr/detail?id=354

[JSR-363] JSR 363 - Units of Measurement 1.0
JSR details http://jcp.org/en/jsr/detail?id=363

[KENNEDY1996] Kennedy, Andrew J. Programming Languages and Dimensions. PhD Thesis, University of Cambridge. Published as Technical Report No. 391, University of Cambridge Computer Laboratory, April 1996.

[MEEP8_OPTIONAL] JSR 361 (MEEP 8) Optionality and Dependencies,
http://docs.oracle.com/javame/config/cldc/opt-pkgs/api/meep/api/doc-files/optionality.html

[MP_METRICS] Eclipse MicroProfile Metrics:
https://microprofile.io/project/eclipse/microprofile-metrics

[MICROMETER_AM] Micrometer Application Monitoring: https://micrometer.io/

[NETFLIX_ARCH] Netflix Archaius Library for configuration management:
https://github.com/Netflix/archaius

[NETFLIX_IEP] Netflix Insight Engineering Platform Components: https://github.com/Netflix/iep

[NIST]	US National Institute of Standards and Technology, International System of Units (SI), Guide for the Use of the International System of Units (SI).
[OSGIM]	OSGi Compendium Measurement, http://www.osgi.org/javadoc/r5/cmpn/org/osgi/util/measurement/package-summary.html
[PCP]	Performance Co-Pilot: http://www.pcp.io/
[PARFAIT]	PCP - Parfait: https://github.com/performancecopilot/parfait
[SOG]	SOG documentation: https://www.eoas.ubc.ca/~sallen/SOG-docs/index.html
[SOG_YAML]	SOG YAML Grammar: https://www.eoas.ubc.ca/~sallen/SOG-docs/SOG-YAML.html#sog-yaml-grammar
[UCUM]	The Unified Code for Units of Measure: Specification
[VECTOR]	Netflix Vector: https://github.com/Netflix/vector
[YAML]	The Official YAML Web Site: http://yaml.org/

Links

- GitHub project site
  https://unitsofmeasurement.github.io
- API JavaDoc
  https://unitsofmeasurement.github.io/unit-api/site/apidocs/index.html or
  https://www.javadoc.io/doc/javax.measure/unit-api/
- API Sources https://github.com/unitsofmeasurement/unit-api, the release tag matching the Maintenance Release 2 of JSR 385 is 2.2