

JSR-275 Specification

Units of Measure

Version 0.9.5 - February 24th, 2010

Content

| | |
|--|----|
| 1. Introduction..... | 3 |
| 1.1 Related work..... | 3 |
| 2. Requirements and Design Goals..... | 5 |
| 2.1 Requirements..... | 5 |
| 2.2 Design Goals..... | 5 |
| 2.3 Source Code and Binary Compatibility..... | 5 |
| 3. Definitions of terms..... | 6 |
| 3.1 Terms used in this specification..... | 6 |
| 3.2 Example..... | 7 |
| 4. Introduction to the API..... | 8 |
| 4.1 Packages and fundamental types..... | 8 |
| 4.2 The Quantity Interface..... | 8 |
| 4.2.1 Quantity instantiation..... | 9 |
| 4.2.2 Quantity as a parameter type..... | 9 |
| 4.3 The Unit Class..... | 9 |
| 4.4 Systems of Units..... | 11 |
| 4.4.1 Metre or Meter?..... | 11 |
| 4.5 Obtaining Unit Instances..... | 12 |
| 4.5.1 Units Obtained From Predefined Unit Constants..... | 12 |
| 4.5.2 Units Obtained From Algebraic Operations..... | 12 |
| 4.5.3 Units Obtained From Unit Symbol Parsing..... | 14 |
| 4.6 Unit Parametrization..... | 16 |
| 4.7 Unit conversions..... | 16 |
| 4.7.1 Creating UnitConverters..... | 17 |
| 5. Supported units..... | 18 |
| 5.1 Supported quantities..... | 18 |
| 6. Integration | 20 |
| 6.1 Existing API..... | 20 |
| 7. Frequently asked questions..... | 21 |
| 8. Addendum..... | 22 |
| 8.1 JavaDoc..... | 22 |
| 9. References..... | 23 |

1. Introduction

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 the unenviable option to either use an inadequate model of measurement or to painstakingly 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-275 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 of predefined units

1.1 Related work

The library presented in this JSR takes its inspiration from some sources:

- *Scientific and Engineering C++* [BART_NACKMAN] presents the implementation strategy used in this JSR, which is to derive every every kind of units 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 can not be transposed to Java, nevertheless their principles still apply.
- *University Corporation for Atmospheric Research* (UCAR)¹ implements a Java units library based on `Unit` and `Converter` types, which are the foundation of this JSR (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 JSR 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. While the language is not in all parts comparable to Java, Unit support is to a large extent. [CUL_QUANT] and [CURL_WHIRL]
- Andrew Kennedy, both with his thesis and works on the topic [KENNEDY1996] as well as recent implementations using languages like F# and others [AKENN]

1 <http://www.unidata.ucar.edu/software/netcdf-java/v4.1/javadocAll/ucar/units/package-summary.html>

- In [FOWLER1996], Martin Fowler analyzed this problem but we needed and implemented a more general model.
- Works on other Object-Oriented languages like Smalltalk which clearly had influence on Java in other areas [ACONCAGUA] or [INGALLS] while similar to other languages some elements like multiple polymorphism are out of scope.
- Contributions came from JSR-108 an earlier JCP approach to Units of Measure in Java [JSR-108]
- While specific to Date and Time only and currently inactive, [JSR-310] is somewhat related and where beneficial to the platform synergies may arise should JSR-310 progress further.

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 of this JSR.

2. Requirements and Design Goals

2.1 Requirements

Core JSR 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.
- Interoperability with code that does not use the unit framework defined in the JSR
- Support for user-defined dimensions, units, quantities and systems of units.
- Support for strict symbol parsing, including:
 - Metric Prefixes (as described in § 5) solely 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 run-time queries on (and printing of) units, measurements, and dimensions. For example, a unit that is ($\text{kg}^1 \cdot \text{m}^1 \cdot \text{s}^{-2}$) 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 run-time.

2.2 Aspirations

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

- Small or no run-time overhead compared with an implementation not using JSR-275.
- Support for the use of more than one unit system simultaneously. For example, a program should be able to simultaneously operate on SI, CGS, and imperial units.
- Compile-time detection of dimension errors when sufficient information is available for a compile-time check.
- Support for fractional exponents, such as $\text{kg}^{3/2}$. Fractional exponents sometimes appear as a partial result on the way to a final value.

2.3 Source Code and Binary Compatibility

Java 6 source code constructs and compilers will be used. Many uses of the proposed package have been discussed on Java ME platforms, such as remote monitoring and sensor equipment. These platforms currently support only limited subsets of the Java 2 APIs. However, Oracle's proposed unification of Java SE and ME opens the possibility that JSR-275 could one day work in a mobile subset or profile. The JSR-275 reference implementation should avoid creating additional barriers to eventual support for the Java ME platform.

3. Definitions of terms

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 example² as the standard unit of length. A similar approach can be taken for monetary quantities, 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 Celsius degrees can be thought as a measurement in Kelvin degrees offset by 273.15 degrees for convenience. A measurement in decibel can be thought 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](#) 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 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.

3.1 Terms used in this specification

The Java classes defined in this specification are not limited in application to physical quantities. The same API can be applied to monetary quantities, 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*”.

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 JSR, 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 sometime considered as a partial³ definition of *Measure* rather than *Quantity*. Nevertheless this JSR uses the “quantity” term for consistency with usage in other frameworks, because the concept of quantity is needed anyway for units parametrization, and for avoiding the introduction of both concepts in the library.

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).

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

³ A more complete definition would include information about the precision of the measurement.

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.

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.

Derived Unit:

A unit formed as a product of exponentiated base units. Some derived units get a special name and symbol for convenience. In the SI system, derived units with special name include *hertz*, *Newton* and *Volt*.

The term 'derived unit' is also used in a looser fashion for the result of a base unit scaled to some factor (for example *kilometre* defined as 1000 metres) 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 and Imperial System.

Prefix:

A leading word that can be applied to a unit to form a decimal multiple or sub-multiple of the unit. Prefixes are primarily used in the Metric system, for example *kilo-* and *centi-*.

3.2 Example

The result of an experience measuring the wavelength of some 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 ”.

4. Introduction to the API

4.1 Packages and fundamental types

This specification comprises two packages:

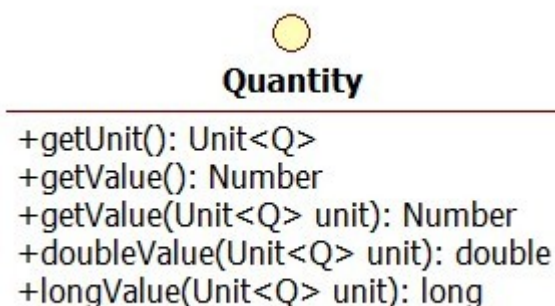
javax.measure.quantity and **javax.measure.unit**

There is no `javax.measure` package⁴. The two main types of this JSR, `Quantity` and `Unit`, are defined in packages of the same name.

The `javax.measure.quantity` package defines the quantity types (e.g. `Mass`, `Length`), while the `javax.measure.unit` package defines the units in which quantities can be expressed (e.g. `gram`, `metre`). Quantities and units work in synergy through use of parametrization features added to the Java language in Java 5.

4.2 The *Quantity* Interface

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



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

It further contains a converter method `getValue(Unit<Q>)` to convert the value to a desired target unit:

```

public Number getValue(Unit<Q> unit) {
    return getUnit().getConverterTo(unit).convert(getValue());
}
  
```

For lower level or performance critical applications (like measuring devices, POS, RFID,...) `Quantity` provides 2 additional methods to convert values directly into a target unit. They are more useful as meta-data converted to the application internal representation (typically a `double` primitive type in some fixed units) before computation or further processing begin. For this purpose, the `Quantity` interface provides the `longValue(Unit<Q>)` and `doubleValue(Unit<Q>)` convenience methods. We encourage developers to prefer them when possible, because they are safer (they force the target

⁴ Similar situations exist in the standard Java library. For example there is a `javax.sound.midi` package and a `javax.sound.sampled` package, but no `javax.sound` packages. See also `javax.security` and `javax.enterprise` packages.

unit to be specified) and avoid significant boiler plate. For example the `doubleValue(Unit<Q>)` method is equivalent to the following code⁵:

```
public double doubleValue(Unit<Q> unit) {
    return getUnit().getConverterTo(unit).convert(getValue().doubleValue());
}
```

With such method, user code like below are much easier to write:

```
Time calculateTravelTime(Length distance, Velocity velocity) {
    double seconds = distance.doubleValue(METRE) /
        velocity.doubleValue(METRE_PER_SECOND);
    return QuantityFactory.getInstance(Time.class).create(seconds, SECOND);
}
```

The more fundamental `getValue()` and `getUnit()` methods are more useful in situations where the `Quantity` type (and consequently the valid units that can be passed to `doubleValue(Unit<Q>)`) are unknown at compile-time, or when the measurement need to be expressed in its original units⁶.

4.2.1 Quantity instantiation

A `QuantityFactory` can be used to generate simple quantities.

```
Time t = QuantityFactory.getInstance(Time.class).create(12, MILLI(SECOND));
Length len = QuantityFactory.getInstance(Length.class).create(34.5, MILES);
```

4.2.2 Quantity as a parameter type

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

```
Thread.wait(long, Unit<Time>); // Unit parametrized with Quantity.
Sensor<Temperature> sensor ... // Generic sensor.

QuantityFactory factory = QuantityFactory.getInstance(Mass.class);
Mass mass = factory.create(180, POUND);
Vector3D<Velocity> aircraftSpeed
    = new Vector3D(12.0, 34.0, -45.5, METRE_PER_SECOND);
```

⁵ Implementations of this JSR are likely to provide more efficient code to their users, for example avoiding the value boxing followed by an un-boxing.

⁶ The original unit gives some indication about the historical period and the instrument. For example a measure of 0.04 metre suggests that the measurement has been done with some instrument designed for metre-scale phenomena, while a measure of 4 cm suggests that the measurement has been done with an instrument better suited to that scale. Ideally such assumptions are irrelevant since every measurement shall be associated with information about its precision. In practice, information are often incomplete and an access to the original data is desirable.

Other example: oceanographers traditionally expressed salinity in ‰ prior 1978, and using the *Practical Salinity Scale* since that time. Conversions are possible between the two systems, but expressing the measures in their original units give sometime useful hints to oceanographer about the measurement.

4.3 The Unit Class

The second (and last) fundamental class with which the user needs to become familiar is `javax.measure.unit.Unit`. The unit class represents a well-known quantity that has been adopted as a standard of measurement. For example “kilometre” and “watt” are units.

| Unit |
|---|
| +ONE: Unit<Dimensionless> |
| <<create>> #Unit() <<create>> +valueOf(CharSequence charSequence): Unit<?> +add(double offset): Unit<Q> +alternate(String symbol): Unit<A> +getSymbol(): String +asType(Class<T> type): Unit<T> +divide(double divisor): Unit<Q> +divide(long divisor): Unit<Q> +divide(Unit<?> that): Unit<Q> +inverse(): Unit<?> +isCompatible(Unit<?> that): boolean +multiply(double factor): Unit<Q> +multiply(long factor): Unit<Q> +multiply(Unit<?> that): Unit<Q> +pow(int n): Unit<?> +root(int n): Unit<?> +toMetric(): Unit<Q> +toString(): String +transform(UnitConverter operation): Unit<Q> |

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 parametrized by a quantity type (§ 4.6).

The dimensions of a unit (`javax.measure.unit.Dimension`) can be retrieved at run-time from the unit class. 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 having no common ancestor other than `Object`.

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

The association of many numerical values to the same unit can be aggregated into container objects such as vectors, columns in a table, remote sensing data, etc. Such constructs outline a means of storing homogeneous measurements. For example, a vector can be defined as a quantity with both a magnitude and a direction, specified by an array of values associated with a shared unit:

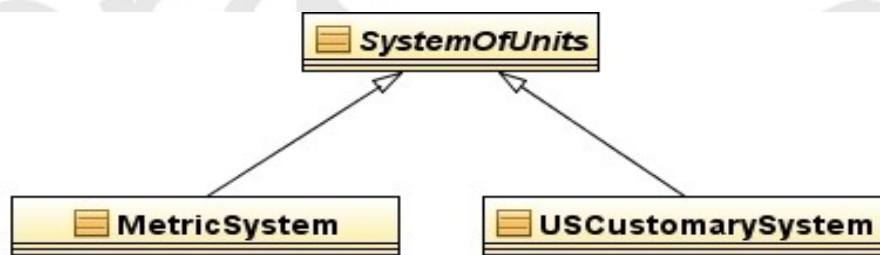
```
public class Vector<Q extends Quantity> implements Quantity<Q> {
    double[] values;
    Unit<Q> unit; // Single unit for all vector components.
    ...
}
```

Associating a single unit with a large set of numerical values is not only more efficient 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 to 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.

4.4 Systems of Units

A *system of units* is a set of units chosen as standards 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 units system is the Metric System, or SI.



Base units in SI include *metre* (m), *kilogram* (kg) and *second* (s). Derived units in SI comprise *square metre* (m²), *watt* (m²·kg·s⁻³), etc. Some derived units have been given special names and symbols for convenience. For example the above-cited *watt* derived unit is represented by the symbol **W**.

In the `MetricSystem`, 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).

Systems of units may share the same units. For example, the Imperial System would have many of the units already defined in `USCustomarySystem` (e.g. `ImperialSystem.FOOT`).

4.4.1 Metre or Meter?

The former is British spelling and the later is U.S. An on-line dictionary of English gives the following explanation: “a metre is a measure and a meter a measuring device”. The BIPM brochure, ISO 31 and Wikipedia use “metre”, while NIST use “meter” (e.g. NIST SP 1038). For this specification, we retained the BIPM “meter” spelling in the text unless explicitly required otherwise. All SI constants names (e.g. `MetricSystem.METRE`) use the BIPM spelling, but for `USCustomarySystem` and systems which may derive from it an alias called `METER` is offered. Similarly for other cases like liter vs. litre. Those unit types are equal.

4.5 Obtaining Unit Instances

Unit instances can be obtained in a number of ways. The easiest way is to use one of the predefined constants provided in various `SystemOfUnits` subclasses like `MetricSystem` (§ 4.5.1). Because the constants are known to the compiler, this approach also provides compile-time checking as discussed in § 4.6. New units can also be created by applying algebraic operations to existing units (§ 4.5.2), or by parsing a unit symbol (§ 4.5.3).

4.5.1 Units Obtained From Predefined Unit Constants

System units can usually be obtained as class members of a system of units. `SystemOfUnits` subclasses are encouraged (but not required) to expose all their system units as static final constants. For example the `MetricSystem` class provides static constants like `METRE` (a basic unit) or `WATT` (a derived unit).

In the particular case of the `MetricSystem` class, a set of static methods is also provided for obtaining multiples or sub-multiples of system units. Method names mirror SI prefix names but are uppercase. Examples⁷:

```
Unit<Length> m = MetricSystem.METRE;
Unit<Length> km = MetricSystem.KILO(MetricSystem.METRE);
Unit<Energy> kw = MetricSystem.KILO(MetricSystem.WATT);
```

Unit instances returned by one of these methods provide type safety through unit parametrization (the `<Length>` and `<Energy>` type parameters). This feature will be discussed more extensively in § 4.6.

4.5.2 Units Obtained From Algebraic Operations

Units can be created dynamically as the result of algebraic operations on existing unit objects. With the exception of the static convenience methods defined in the `MetricSystem` (§ 4.5.1), all operations are defined as member methods of the `Unit` class. The `MetricSystem` methods are omitted from discussion below⁸.

Operands:

All binary operations expect `(Unit, Unit)` or `(Unit, n)` operands where n stands for a real value (sometime 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`”, “`m2`” or “`m·1000`”, where the operations represented by “`/`”, “`·`” and raising to a power correspond to the Java methods `divide(Unit)`, `multiply(Unit)` 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:

- **Result has the 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

⁷ We keep the `MetricSystem` class name in those examples for clarity. For many applications, usage of static imports will make the reading easier, as in `KILO(METRE)`.

⁸ All static methods defined in the `MetricSystem` class can be implemented in terms of calls to `Unit` member methods. For example `MetricSystem.KILO(Unit)` can be mapped to `Unit.times(1000)`

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. The algebraic functions appear in bold characters. 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 `MetricSystem` class, which is assumed implicit 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 double imprecision (the internal representation of 2.54 is actually something like 2.540000000000000036...)

- **Result has a 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 in the `Unit` class.

| Result with same dimension | Result with different dimension |
|--|---|
| <p><i>Binary operations:</i></p> <ul style="list-style-type: none"> • add (double) or (long) • multiply (double) or (long) • divide (double) or (long) • compound (Unit) | <p><i>Binary operations:</i></p> <ul style="list-style-type: none"> • root (int) • pow (int) • multiply (Unit) • divide (Unit) <p><i>Unary operations:</i></p> <ul style="list-style-type: none"> • inverse() |



Instead of `add()`, `plus()` is also a proposed method name. It has been reported that some users expect “`A.add(B)`” to perform an addition in place (i.e. the new value replace the old value into A), while “`A.plus(B)`” suggests the same behavior as the `+` operator.

Futhermore, the 'plus' operation over `Unit` is semantically different from the `add` operation between numbers, as it does not mean that something is added to the unit (which would not make sense) but rather that the unit is offset by the specified amount. Therefore **`offset()`** like proposed by precursor JSR-108 seems the best option to most EG members for this case.

4.5.3 Units Obtained From Unit 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 the `UnitFormat` class. This class 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 “**kW**” 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)`.

`UnitFormat` is defined as a subclass of `java.text.Format` in order to facilitate its usage with existing libraries like `javax.swing.JFormattedTextField`. Like most format classes, it contains a set of `getInstance(...)` static methods. They are detailed together with the rest of the `UnitFormat` API. Users can get `Unit` objects directly from a `UnitFormat` instance – this approach provides the greatest flexibility – or from the `Unit.valueOf(CharSequence)` convenience static method, which delegates to the standard `UnitFormat` instance. This standard format recognizes all units enumerated in the SI brochure and U.S. customary units (§ 5).

Examples:

```
Unit<Length> metre = Unit.valueOf("m").asType(Length.class);
Unit<Length> feet = Unit.valueOf("ft").asType(Length.class);
Unit<Energy> kilojoule = Unit.valueOf("kJ").asType(Energy.class);
Unit<Force> newton = Unit.valueOf("m.kg/s2").asType(Force.class);
```

The `asType(Class)` method checks that the return value of the `valueOf(...)` method, `Unit<?>`, is of the proper quantity type (e.g. `Unit<Length>`). Quantity type checking is further elaborated in § 4.6.

4.6 Unit Parametrization

Units are always checked for compatibility at run-time prior to any operation. For example, any attempt to convert a quantity from kilogram units to metre units will result in a `ConversionException` being thrown. Rigorous run-time 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 run-time checks is not always significant (§ 4.7).

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

There are two ways to obtain a parametrized unit:

- Assignment from one of the predefined constants, such as those defined in the `MetricSystem` 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 (`times`) and division (`divide`) 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 can not 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 a “unchecked cast” warning for such code.

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

As of Java 5, checks can not 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    = Unit.valueOf("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` 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.

4.7 Unit conversions

Conversions involve two steps: 1) obtain a `UnitConverter` object for a given pair of source and target units, and 2) use it to convert an 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 c = sourceUnit.getConverterTo(targetUnit);
double length1 = 4.0;
double length2 = 6.0;
length1 = c.convert(length1);
length2 = c.convert(length2);
```

This example illustrates the advantages of having a `UnitConverter` class as opposed to simply including a method in the `Unit` class 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` class encapsulates the result of this operation. Once available, it can be applied efficiently on a large number of values. Different implementations exist for different kinds of unit scales (identity, linear, logarithmic, *etc.*).

|  UnitConverter |
|--|
| <p style="text-align: center;"><i>Operations</i></p> <pre> protected UnitConverter() public UnitConverter inverse() public double convert(double value) public BigDecimal convert(BigDecimal value, MathContext ctx) public boolean equals(Object cvtr) public int hashCode() public UnitConverter concatenate(UnitConverter converter) public boolean isLinear() </pre> |

4.7.1 Creating UnitConverters

There is no factory for creating `UnitConverter` instances directly. Converters of different kinds are created indirectly by the various `Unit.getConverterTo` methods. For example the two following lines create the same `UnitConverter`, but the second one is the expected use case since JSR-275 is about unit conversions (as opposed to a general mathematical library):

```

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

```

5. Supported units

The out-of-the-box units database shall include support for the units defined in the following documents:

- BIPM units (same as ISO 1000), including:
 - [Base units](#)
 - [Dimensionless derived units](#)
 - [Derived units with special names](#)
 - [Non-SI units accepted for use with SI](#)
 - [SI prefixes from \$10^{-24}\$ to \$10^{+24}\$](#) .
- ISO 31 including:
 - ISO 31-1: Space and time
 - ISO 31-2: Periodic and related phenomena
 - ISO 31-3: Mechanics
 - ISO 31-4: Heat
 - ISO 31-5: Electricity and magnetism
 - ISO 31-6: Light and related electromagnetic radiations
 - ISO 31-7: Acoustics
 - ISO 31-8: Physical chemistry and molecular physics
 - ISO 31-9: Atomic and nuclear physics
 - ISO 31-10: Nuclear reactions and ionizing radiations
 - ISO 31-11: Mathematical signs and symbols for use in the physical sciences and technology
 - ISO 31-12: Characteristic numbers
 - ISO 31-13: Solid state physics
- NIST
 - NIST Handbook 44 - 2002 Edition; Specifications, Tolerances, And Other Technical Requirements for Weighing and Measuring Devices.
 - [Appendix C - General Tables of Units of Measurement](#)
 - [Federal Standard 376B](#)
 - [NIST Special Publication 811](#) - Guide for the Use of the International System of Units (SI); 1995. (Specifically units defined in appendix B8)

5.1 Supported quantities

Quantity subtypes are defined only for a subset of the supported units. The table below lists the minimal set of quantity types that compliant implementations shall define, together with the name of the `Unit` constant declared in the `MetricSystem` class.

This table can be extended with user-defined quantities. However for any quantity not included in this JSR, interoperability between the quantities defined in different libraries is reduced. For example if two independent libraries named “A” and “B” define a Torque quantity in their own respective packages, it is not possible to use a Torque instance of library A in places where a Torque instance of library B is expected. Their associated `Unit<Torque>` instances can not be interchanged neither, 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 |
|------------------------|--------------------------|------------------|
| Acceleration | METRES_PER_SQUARE_SECOND | m/s ² |
| AmountOfSubstance | MOLE | mol |
| Angle | RADIAN | rad |
| Area | SQUARE_METRE | m ² |
| CatalyticActivity | KATAL | kat |
| DataAmount | BIT | bit |
| ElectricCapacitance | FARAD | F |
| ElectricCharge | COULOMB | C |
| ElectricConductance | SIEMENS | S |
| ElectricCurrent | AMPERE | A |
| ElectricInductance | HENRY | H |
| ElectricPotential | VOLT | V |
| ElectricResistance | OHM | Ω |
| Energy | JOULE | J |
| Force | NEWTON | N |
| Frequency | HERTZ | Hz |
| Illuminance | LUX | lx |
| Length | METRE | m |
| LuminousFlux | LUMEN | lm |
| LuminousIntensity | CANDELA | cd |
| MagneticFlux | WEBER | Wb |
| MagneticFluxDensity | TESLA | T |
| MagnetomotiveForce | AMPERE_TURN | At |
| Mass | KILOGRAM | kg |
| Power | WATT | W |
| Pressure | PASCAL | Pa |
| RadiationDoseAbsorbed | GRAY | Gy |
| RadiationDoseEffective | SIEVERT | Sv |
| RadioactiveActivity | BECQUEREL | Bq |
| SolidAngle | STERADIAN | sr |
| Temperature | KELVIN | K |
| Time | SECOND | s |
| Velocity | METRES_PER_SECOND | m/s |
| Volume | CUBIC_METRE | m ³ |

6. Integration

6.1 Existing API

6.1.1 JDK

Java 5 defines a `java.util.concurrent.TimeUnit` enum which overlaps the purpose of the `Unit<Time>` construct defined in this JSR. The two constructs can be made interoperable by the addition of the following methods in the `TimeUnit` enum:

- `public Unit<Time> toUnit();`
- `public static TimeUnit valueOf(Unit<Time> unit)
throws IllegalArgumentException;`

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

We do not recommend any change in the current API using `TimeUnit` for the following reasons:

- Because `Unit<Time>` can not be retrofitted as a subclass of `TimeUnit`, supporting `Unit<Time>` would require a duplication of every API currently working with `TimeUnit`.
- Conversions between two `Unit<Time>` are slower than conversions between two `TimeUnit`, because `Unit<?>` is a more generic class which need to check the dimension of the source and target units. In contrast `TimeUnit` doesn't need to perform such check since it is restricted to the time dimension.
- Using the `Time` quantity would be yet slower because of the number auto-boxing.
- Another proposed JCP standard, [JSR-310] – Time & Date deals specifically with the detailed modeling of time aimed to replace types currently in the `java.util` Package like `Date` or `Calendar`. Its current stage contain several types one of which named `PeriodUnit`. The Curl language [CUL_QUANT] being a key inspiration here defines `Date & Time` also side by side with `Units & Quantities`.

`Unit<Time>` and `TimeUnit` serve different purpose. 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.Unit` is a generic and extensible framework used in context where performance is less critical, but the task wider.

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 be deserve a special status because of that.

6.1.2 ICU4J

This open library [ICU4J] driven and developed by an industry group of companies like IBM, Oracle, Adobe, Apple or Google has created at least two main libraries. One for C which is not directly relevant here. One for Java 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-275. 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-275 would also allow its adoption by future ICU versions. Minor differences in the Quantity/Measure exist like its value method called `getNumber()`, again other approaches “closer” to this JSR like 310 (see below) call it `getAmount()` so we won't elaborate further about them here.

6.2 Other JSR

As mentioned, [JSR-310] the JCP Date & Time proposal shows a few similarities we want to list without going into specific detail about them. Since the JSR is currently inactive and according to EG members planning for EDR this year, so timing and possible launch may allow synergies where appropriate.

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 JSR, there are 3 main points of interest:

- **PeriodUnit**
Not only called a unit but also very similar in aim and name to the enum Java Concurrency API.
- **PeriodField**
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 PeriodUnit unit;
```

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

- **Duration**
Interconnected with `PeriodUnit` this is not related to `PeriodField` 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 sub-classes 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 JSRs. After all people who pick a date using a calendar widget usually wish to do something with it.

Calculate how many TB of data their customers downloaded, the remaining quota, 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.

7. Frequently asked questions

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

The first concept from which all others are derived is Quantity (§ 3). Quantities are suitable to parametrize units because they answer an important question at compile time: What kind of quantity a unit represents. The standard parametrization 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 parametrization 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 kind of quantity.
- Users will work primarily with `Quantity` and `Unit` objects. The need to work with `Dimension` objects is less common. If the units were parametrized 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 JSR use more Enum or Annotation types?

- Enums
The first thought for one of the few strictly limited parts of the JSR, the dimension was in fact the **enum** type. It is final and is defined only by a small set of well-known elements. The fact, that NONE (Dimensionless) has no symbol but a default unit made it harder to deal with enums and their default constructors. A default yet extensible model and dimensional calculations made it even harder..

Enums are used in other areas, e.g. the internals of unit parsing and formatting. In most other parts of the API we need to keep it open and flexible towards extensions. Which a final type like enum wouldn't work for. In cases where a quantity implementation shall be final and values would be a selection of choices rather than a numeric value, using enums to implement such quantity seems tempting. Using interfaces to interconnect different but related sets of enums is suggested by . It works in various practical frameworks EG members worked on in other areas. And looking at restrictions some already final JSRs are bound to by their use of enums lets one hope its next releases find alternatives here, too.

The following example of how to use unit declarations instead of primitive or simple Number objects was based on code from the Enums chapter in [EFFECTIVEJAVA] (p. 150ff)

```
import javax.measure.quantity.Length;
import javax.measure.quantity.Mass;
import static javax.measure.unit.USCustomarySystem.METER;
import static javax.measure.unit.MetricSystem.KILOGRAM;

public enum Planet {
    MERCURY (KILOGRAM.multiply(3.303e+23), METER.multiply(2.4397e6)),
    VENUS   (KILOGRAM.multiply(4.869e+24), METER.multiply(6.0518e6)),
    EARTH   (KILOGRAM.multiply(5.976e+24), METER.multiply(6.37814e6)),
    MARS    (KILOGRAM.multiply(6.421e+23), METER.multiply(3.3972e6)),
    JUPITER (KILOGRAM.multiply(1.9e+27),   METER.multiply(7.1492e7)),
    SATURN  (KILOGRAM.multiply(5.688e+26), METER.multiply(6.0268e7)),
    URANUS  (KILOGRAM.multiply(8.686e+25), METER.multiply(2.5559e7)),
    NEPTUNE (KILOGRAM.multiply(1.024e+26), METER.multiply(2.4746e7)),
    PLUTO   (KILOGRAM.multiply(1.27e+22),  METER.multiply(1.137e6));

    private final Unit<Mass> mass; // in kilograms
    private final Unit<Length> radius; // in meters
    Planet(Unit<Mass> mass, Unit<Length> radius) {
        this.mass = mass;
        this.radius = radius;
    }
    public Unit<Mass> mass() { return mass; }
    public Unit<Length> radius() { return radius; }
    [...]
}
```

- Annotations

Also mentioned by [EFFECTIVEJAVA] is, that defining new types is better done by interfaces than annotations. Again using annotation-based JSRs like 330 (DI) or Bean Validation together with JSR-275 seems an almost natural 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 JSR as such defines the types for this first.

8. Addendum

8.1 *JavaDoc*

API Docs at GeoAPI Hudson Site:

<http://hudson.geomatys.com/job/JSR-275/ws/jsr-275/target/site/apidocs/index.html>

9. References

- [BIPM] *Bureau International des Poids et Mesures*
[Brochure on the International System of Units](#)
- [ISO] *International Organization for Standardization*
- [ISO 31](#) (Quantities and units)
 - ISO 1000
 - [ISO 10303 STEP Part 41](#) .
 - [STEP Part 41 EXPRESS Schema](#) .
- [NIST] *US National Institute of Standards and Technology*
- [International System of Units \(SI\)](#)
 - [Guide for the Use of the International System of Units \(SI\)](#)
- [UCUM] The Unified Code for Units of Measure: [Full Specification](#)
- [BART_NACKMAN] Barton J.J and Nackman L.R., 1994. *Scientific and engineering C++ – an introduction with advanced techniques and examples*. Addison-Wesley.
- [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.
- [DELFT1999] Delft, A., 1999 – A Java Extension With Support for Dimensions.
Software - Practice and Experience 29(7).
- [JSCIENCE] JScience – Java Library for the Advancement of Sciences: [Home Page](#)
- [JSR-108] JCP proposal JSR 108, Source Code at
<http://jsr-108.sourceforge.net/javadoc/javax/units/Unit.html>
JSR proposal at <http://jcp.org/en/jsr/detail?id=108>
- [JSR-310] Date & Time JSR: proposal at <http://jcp.org/en/jsr/detail?id=310>
- [ICU4J] ICU4J see <http://icu-project.org>
- [CURL_QUANT] [Curl – Quantities and Units](#)
- [CURL_WHIRL] [Taking Curl for a Whirl](#)
- [EFFECTIVEJAVA] Bloch, Josh, Effective Java Second Edition
<http://java.sun.com/docs/books/effective/>
- [FOWLER1996] Fowler, M. Analysis Patterns: Reusable Object Models.
Addison-Wesley, Reading, MA, 1996
- [INGALLS] Ingalls, D. A simple technique for handling multiple polymorphism. ACM
SIGPLAN Notices, 21(11):347—349, Nov. 1986
- [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