

# **JSR-275 Specification**

## **Measures and Units**

**Version 0.9.4 - December 04, 2009**

## Content

1. Introduction.....	3
2. Requirements and Design Goals.....	4
2.1 Requirements.....	4
2.2 Design Goals.....	4
2.3 Source code and binary compatibility.....	4
3. Definitions of terms.....	5
3.1 Terms used in this specification.....	5
3.2 Example.....	6
4. Introduction to the API.....	7
4.1 Fundamental types.....	7
4.2 Systems of units.....	9
4.3 Obtaining Unit Instances.....	10
4.3.1 From predefined constants.....	11
4.3.2 From algebraic operations.....	12
4.3.3 From unit symbol.....	13
4.4 Unit Parametrization.....	15
4.5 Units conversions.....	16
5. Detailed Specification (API).....	18
6. Supported units.....	235
7. Frequently asked questions.....	236
8. References.....	237

# 1. Introduction

Java developers who work with physical quantities (such as developers in the scientific, engineering, medical, and manufacturing domains) need to be able to handle measurements of these quantities in their programs. Inadequate models of physical measurements can lead to significant programmatic errors. In particular, the practice of modeling a measure as a simple number with no regard to the units it represents creates fragile code. Another developer or another part of the code may misinterpret the number as representing a different unit of measurement. For example, it may be unclear whether a person's weight is expressed in pounds, kilograms, or stones.

Developers must either use inadequate models of measurements, or must create their own solutions. A common solution can be safer and can save development time for domain-specific work. This JSR proposes to establish safe and useful methods for modeling physical quantities.

JSR-275 specifies one or more Java packages for the programmatic handling of physical quantities and their expression as numbers of units. The specification includes:

- Interfaces and abstract classes with methods for unit operations:
  - Checking of unit compatibility
  - Expression of measurement in various units
  - Arithmetic operations on units
- Concrete classes implementing the standard types of units (such as base and derived units) and unit conversions.
- Classes for parsing and formatting textual units representations.
- A database of predefined units.

## 2. Requirements and Design Goals

### 2.1 Requirements

The following features are considered critical to the success of the JSR.

- No Java Virtual Machine changes are allowed.
- Do no numerical harm. In essence, whatever algorithm the user intended to code should execute as intended.
- Should be able to inter-operate with code that does not understand units.
- Must support user-defined dimensions, units, quantities and systems of units.
- Strict symbols parsing for formatting:
  - Prefixes (as described in § 6) shall only be supported for use with SI units.
- Support run-time queries on (and printing of) units, measurements, and dimensions.

The last requirement implies that it must be possible to make the dimensions, their powers, and their names manifest at run-time. For example, a unit that is ( $\text{kg}^1 \cdot \text{m}^1 \cdot \text{s}^{-2}$ ) should be able to manifest in some version of any or all of the system (SI, in this case), the dimensions ( $\text{kg}$ ,  $\text{m}$ ,  $\text{s}$ ), the exponents of those dimensions (1, 1, -2), and/or the name of the unit (if there is a defined name).

### 2.2 Design Goals

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

- Small or no run-time overhead, compared with an implementation that doesn't use units.
- It should be possible to write "generic" numeric operations that compute correct results without regard to the user's choice of system, unit or quantity, while still remaining type-safe in terms of system, unit and quantity. A sample use-case would be a type-safe Newton's Method Square Root implementation.
- It should be possible to write programs that use more than one unit system simultaneously. For example, a program should be able to operate on SI, CGS, and imperial units, all simultaneously.
- Dimension errors shall be detected at compile time, not deferred to run time, when sufficient information allows a compile-time check.
- Fractional values for dimensions shall be allowed, for example  $\text{kg}^{3/2}$ . Fractional exponents sometime appear as a partial result on the way to a final value.

### 2.3 Source code and binary compatibility

Java 5 source code constructs and compilers will be used. Many uses of the proposed package have been discussed on J2ME platforms such as remote monitoring and sensor equipment. These platforms only support limited subsets of the Java 2 APIs. The JSR reference implementation should not stray from these areas which would make possible the creation of a binary jar file for the J2ME platform.

## 3. Definitions of terms

The measurement of a physical quantity is the process of estimating the ratio of its magnitude to an other quantity of the same kind, which we take for unity. For example “5 meters” is an estimate of an object's length relative to an other length, the *meter*, which we adopt in this example<sup>1</sup> as the standard unit of length. A similar approach can be taken for monetary quantities, providing that a monetary unit is properly defined.

The measurement term is also used in a looser fashion to refer to any process of assigning numbers to represent some sense of degree in an entity. For example the [Mohs scale of mineral hardness](#) characterizes the scratch resistance of various minerals through the ability of a harder material to scratch a softer material. This scale states that quartz is harder than calcite, but doesn't say how much harder. It is a somewhat arbitrary, strictly *ordinal* scale. Quantities expressed according such scales are comparable but not additive.

A measurement can be an estimation or 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

Java classes defined in this specification are not limited in scope to physical quantities. They can be applied to monetary quantities using the same API as the one for physical quantities. Users can also define their own scale, which may be ordinal. 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 properties or attributes of thing. Mass, time, distance, heat, and angular separation are among the familiar examples of quantitative properties.

- No unit is needed to express a quantity, nor to do arithmetic on one. Alice can quantify the mass of her shoe by handling it. 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.

#### *Measurable*

A quantity for which given a compatible unit, the magnitude stated in this unit can be retrieved.

- Measurable are not always scalar quantities; they can be vectors, data sets or any thing for which a meaningful magnitude can be determined. For example, the magnitude of a velocity vector could be its norm, the magnitude of a collection of data samples could be the average value.

#### *Measure*

The result of a measurement, *expressed* as the combination of a numerical value and a unit.

- A measure is a scalar magnitude (in the sense of amount, not the sense of magnitude versus phase). Thus the mass of Alice's shoe is a magnitude.
- Measures represent exact or approximate quantities.
- Whereas the quantity being measured has infinite precision (the mass of Alice's shoe is exact) the measure may be limited by the precision of an IEEE floating point, or by the resolution of the mass scale.

---

<sup>1</sup> This specification do not mandate the use of any particular quantity as standard unit. However, we expect SI to be the standard system of units for most applications.

**Dimension**

One of the fundamental aspects of quantities, such as length ( $L$ ), mass ( $M$ ), time ( $T$ ), or combination thereof ( $ML/T^2$  dimension of force). The notion of dimension expresses a property without any concept of magnitude. Thus 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; and it follows that these 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 can be created from some quantity taken as reference. For example, the foot unit corresponds to a quantity of 0.3048 meters. Regardless of how it is created, a unit can be expressed as a quantity of other units with the same dimension.

**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 *meter*, *kilogram* and *second*), from which all other SI units are derived.

**Derived Unit**

A unit formed as a product of powers of the 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*.

**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 SI system, which includes *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:

- $\lambda$  is the symbol for the physical quantity (*wavelength*)
- **nm** is the symbol for the physical unit (*nanometer*), 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 nanometers.

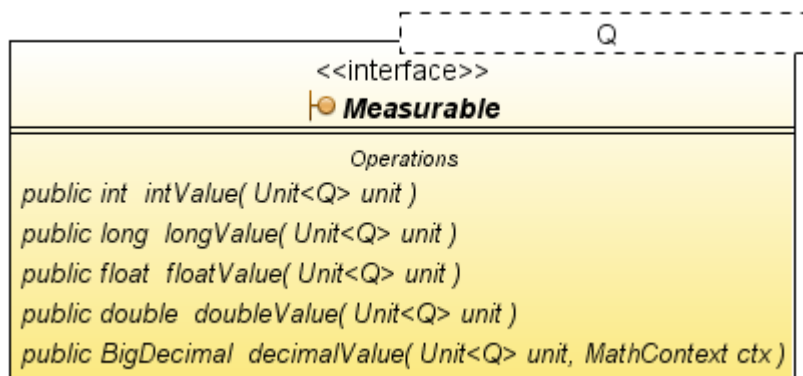
The dimension of length is typically represented by the letter  $L$  in dimensional analysis. However, this dimension does not appear explicitly. Instead, it can be inferred from the quantity “ $\lambda$ ” or from the unit “**m**”.

## 4. Introduction to the API

For a detailed API description see chapter 5.

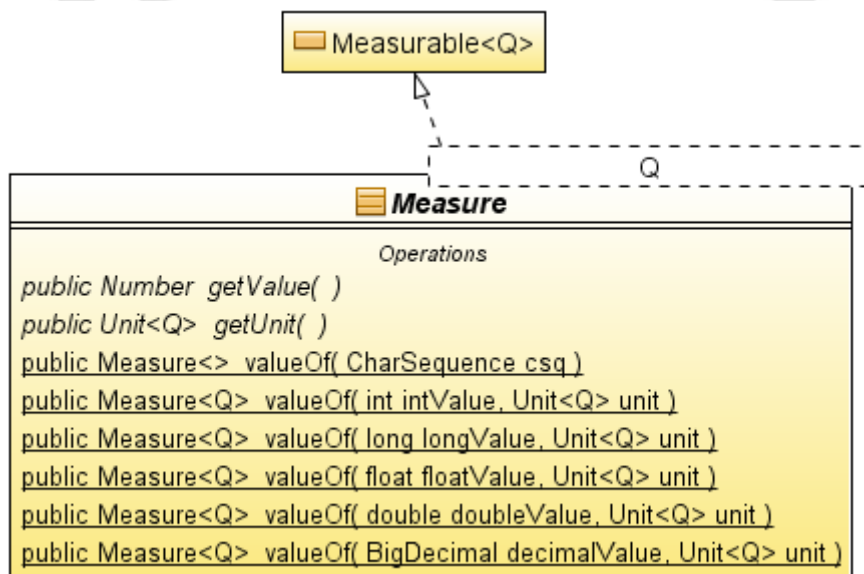
### 4.1 Fundamental types

At the root of this specification is the `javax.measure.Measurable` interface identifying quantities capable of being measured.



Methods of this interface are similar to the ones in `java.lang.Number` except that a unit has to be specified in order to retrieve the desired primitive type. Also in case of overflow, an exception is raised rather than a silent truncation being performed (safety critical).

The second important class from the `javax.measure` package is the abstract class `Measure` representing the result of a scalar measurement stated in a known unit.



For performance reasons, the `Measure` class provides specialized factory methods for primitive types values (avoids auto-boxing).

This last class is mostly for convenience as it allows for the creation of simple `Measurable` instances from scalar decimal representations, primitive types or `BigDecimal` instances.

```

import static javax.measure.unit.SI.*;
import static javax.measure.unit.SI.MetricPrefix.*;
...

// Speed of Light (exact from decimal representation)
Measure<Velocity> c = Measure.valueOf("299792458 m/s").asType(Velocity.class);

// Example of temperature (double approximation).
Measure<Temperature> boilingPoint = Measure.valueOf(373.15, KELVIN);

// Example of duration (exact from int value).
Measure<Duration> delay = Measure.valueOf(200, MILLI(SECOND));

// Example of frequency (arbitrary precision from BigDecimal).
Measure<Frequency> resonance = Measure.valueOf(
    BigDecimal.valueOf("34.55", MathContext.DECIMAL128), GIGA(HERTZ));

```

Finally the last important type the user needs to know is `javax.measure.unit.Unit`, the class representing a determined quantity adopted as a standard of measurement. For example “kilometer” and “watt” are units, but “5 kilometers” is not (at least in the international system of units). It is rather a measure.

Units and measure are defined in terms of each other. While `Unit` could be defined as a subtype of `Measure`, in the sense of “is a kind of”, this specification avoid such relationship in order to discourage the abuse of `Unit` as a general-purpose implementation of `Measure`.

The classes `Measurable`, `Measure` and `Unit` are all parametrized by their quantity type (§ 4.4). Quantity types (such as Mass, Duration) are derived from `javax.measure.quantity.Quantity`. The dimensions of a unit (`javax.measure.unit.Dimension`) can be retrieved from the unit itself (at run-time) or be inferred from its quantity type (at compile time).

**Example:** “centimeter” is a unit. “2.54 centimeters” is a measure of length (quantity), but this measure can be used as the definition of a new length unit called “inch”. “1 inch” is a measure equivalent to 2.54 centimeters. Lets emphasize that “1 inch” (the measure) and “inch” (the unit) are not synonymous. In this specification, they are represented by two distinct types with no common ancestor other than `Object`.

All numerical values in a program 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 measure can be created directly through the `Measure` class.

Many numerical values associated to the same unit can form other kinds of 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 a magnitude and a direction, specified by components all having the same unit:

```

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

```

Associating a single unit to a large set of numerical values has obvious advantages from a storage space point of view. But it is also advantageous on a performance point of view, since the conversion formula to another unit needs to be determined only once for a whole vector. A single unit can be associated to yet more complex objects, like matrices or collections.

`Measure` and `Unit` instances are immutable and thus thread-safe. They are safe for use as static constants (final). Immutability is an essential property of `Unit` since the same instances are typically

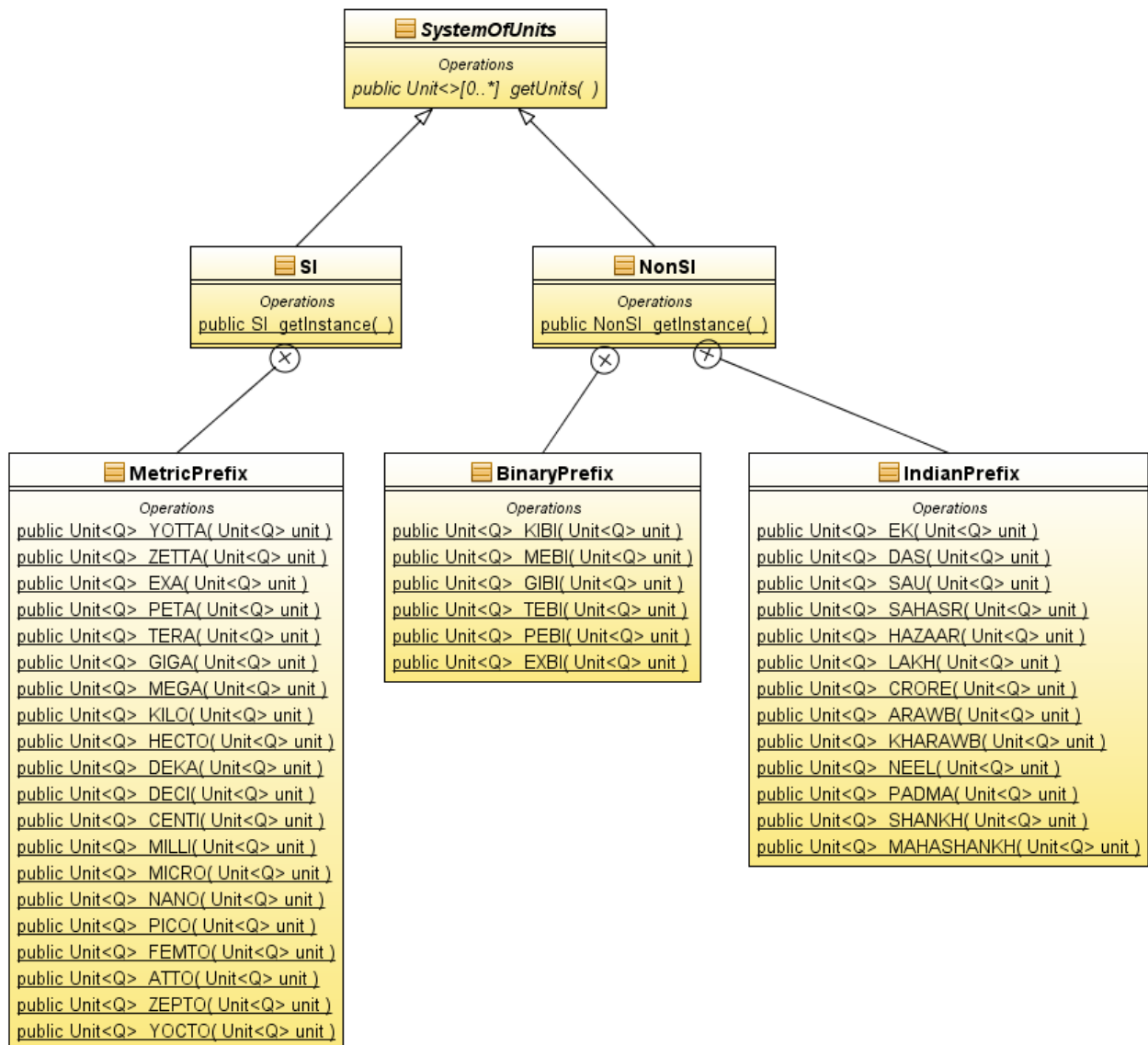
referenced by a large number of quantity objects, and the cost of cloning each unit (*defensively copying*) would be prohibitive.

### 4.2 Systems of units

A *system of units* is a set of units chosen as standards for specifying measurements. It contains a small set of well-defined units, called *base units*, which by convention are regarded as dimensionally independent of other base units. It contains also a larger set of *derived unit* formed as products of powers of the base units. A widely used units system is SI (*Système International*).

Base units in SI comprise *meter* (m), *kilogram* (kg) and *second* (s). Derived units in SI comprise *square meter* (m<sup>2</sup>), *watt* (m<sup>2</sup>·kg·s<sup>-3</sup>), 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*.

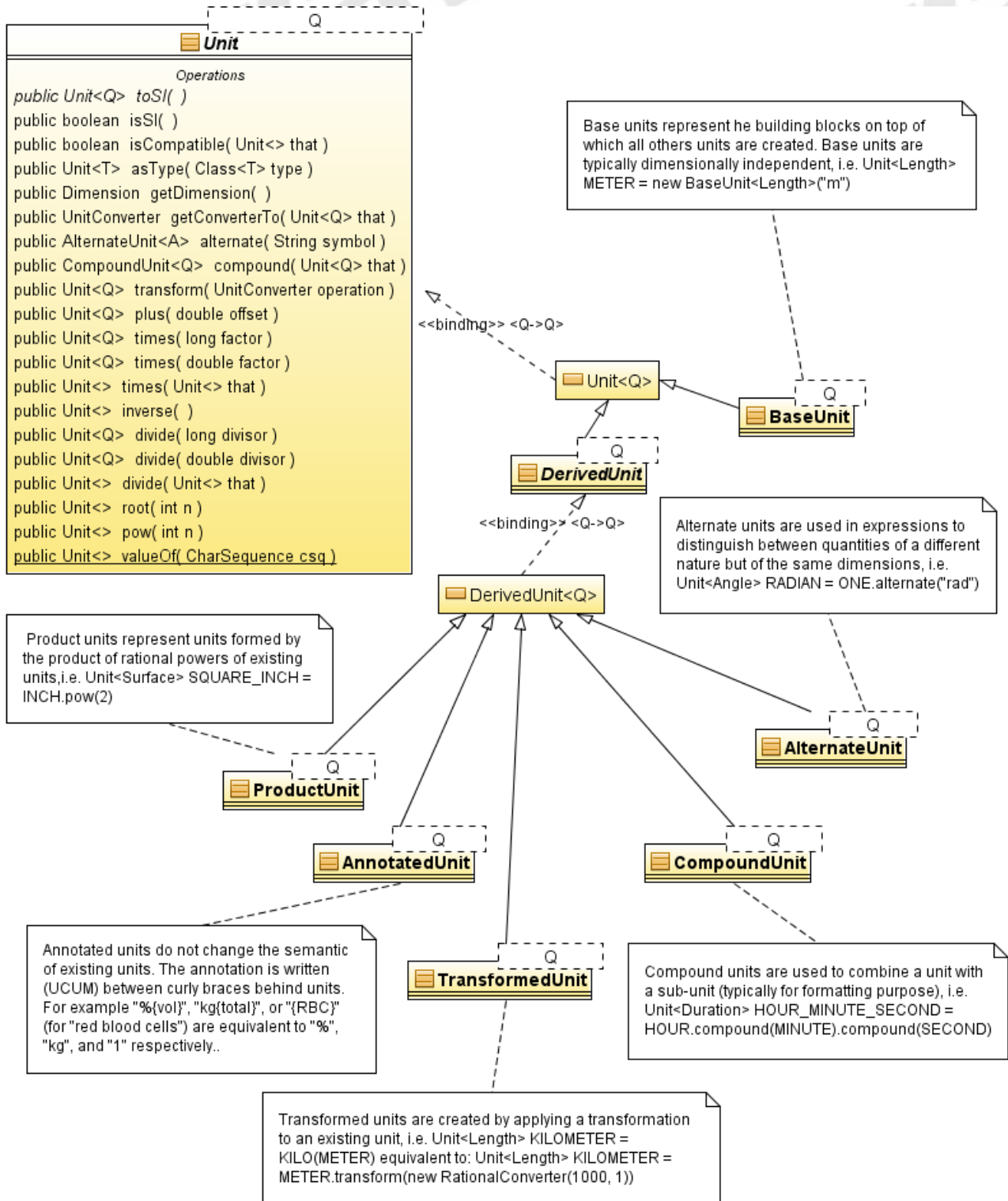
In SI, the base and derived units form a *coherent* set of units, where coherent means that those units are mutually related by rules of multiplication and division with no numerical factor other than 1 (BIPM, 1998).



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

### 4.3 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 **SI** (§ 4.3.1). Because the constants are known to the compiler, this approach also provides compile-time checking as discussed in § 4.4. New units can also be created by applying algebraic operations on existing units (§ 4.3.2), or by parsing a unit symbol (§ 4.3.3).



### 4.3.1 From predefined 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 **SI** class provides such static constants like **METRE** (a basic unit) or **WATT** (a derived unit).



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 “meter” american spelling in the text but we use the BIPM spelling for SI constants names (e.g. SI.METRE, NonSI.LITRE; but UCUM.METER).

In the particular case of the **SI** class, a set of static methods is also provided for obtaining multiples or sub-multiples of system units. All those methods are named as SI prefix names changed to upper case. Examples<sup>2</sup>:

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

**Unit** instances obtained that way provide type safety through unit parametrization (the <Length> and <Energy> declarations above). This feature will be discussed more extensively in § 4.4.

### 4.3.2 From algebraic operations

Units can be created dynamically from existing units by the use of algebraic operations. All operations are defined as member methods of the **Unit** class, except the static convenience methods defined in the **SI** class (§ 4.3.1). Those **SI** methods are omitted from discussion below<sup>3</sup>.

#### **Operands:**

All binary operations expect (**Unit**, **Unit**) or (**Unit**, *n*) operands where *n* stands for an integer value. The first operand is always the **Unit** instance on which the Java method is invoked.

**Example:** lets “m” and “s” be two instances of **Unit**. Some valid operations are “m/s”, “m·s”, “m<sup>2</sup>” or “m·1000” where “/”, “.” and raising to a power maps to Java methods **divide(Unit)**, **times(Unit)** and **power(int)** invoked on the **Unit** instance named “m”.

#### **Result:**

All operations – both unary and binary – return a new or an existing instance of **Unit**. All operations can be grouped in two categories according the dimension of the returned unit, compared to the dimension of the first **Unit** operand:

- **Result** has the same dimension as the operand. Those operations are used for the definition of new units as existing units scaled to some factors, or as existing units with the scale translated 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 centimeters, or equivalently as the centimeter

<sup>2</sup> We keep the **SI** class name in those examples for clarity. For many applications, usage of static imports will make the reading easier, as in **KILO(METRE)**.

<sup>3</sup> All static methods defined in the **SI** class can be implemented in terms of calls to **Unit** member methods. For example **SI.KILO(Unit)** can be mapped to **Unit.times(1000)**

unit scaled by a factor of 2.54.

- Result has a different dimension than the operand. Those operations are used for derivation of new units from existing one (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 table summarizes the algebraic operations provided in the `Unit` class.

<b>Result with same dimension</b>	<b>Result with different dimension</b>
<p><i>Binary operations:</i></p> <ul style="list-style-type: none"> <li>• <b>plus</b> (double) or (long)</li> <li>• <b>times</b> (double) or (long)</li> <li>• <b>divide</b> (double) or (long)</li> <li>• <b>compound</b> (Unit)</li> </ul>	<p><i>Binary operations:</i></p> <ul style="list-style-type: none"> <li>• <b>root</b> (int)</li> <li>• <b>power</b> (int)</li> <li>• <b>times</b> (Unit)</li> <li>• <b>divide</b> (Unit)</li> </ul> <p><i>Unary operations:</i></p> <ul style="list-style-type: none"> <li>• <b>inverse()</b></li> </ul>



An other proposal for method names was `add` and `multiply`, which are more common than `plus` and `times` in JDK and Java3D libraries. However 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 than the `+` operator.

### *Example for the first case*

The following examples define the inch unit as 2.54 centimeters. This unit is defined in three different ways, which are all equivalent up to rounding error. The algebraic functions appear in bold characters. Note that `CENTI(x)` is basically a convenience method for `x.divide(100)` with prefix symbol management added. The `CENTI` and `METRE` members are defined in the `SI` 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 (meter).

- `Unit<Length> INCH = CENTI(METRE).times(2.54);`
- `Unit<Length> INCH = METRE.times(0.01).times(2.54);`
- `Unit<Length> INCH = METRE.times(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...)

### *Example for the second case*

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);`

### 4.3.3 From unit symbol

Units can be created dynamically from their string representation – usually their unit symbol, but 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:

- Maintains associations between an arbitrary amount 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.
- Maps prefix symbols to the corresponding scaling methods, if applicable. For example a **UnitFormat** instance for SI symbols should map the “**kW**” string to the **KILO(WATT)** unit.
- Maps operator symbols to the corresponding arithmetic methods. For example a **UnitFormat** instance for **SI** symbols should map the “**m/s**” string to the **METRE.divide(SECOND)** unit.

**UnitFormat** is defined as a **java.text.Format** subclass 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 (UCUM<sup>4</sup> **UnitFormat** instance. This standard format recognizes all SI and non-SI units enumerated in the SI brochure and U.S. customary units (§ 6). 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 **valueOf(...)** return value **Unit<?>** is of the proper parametrized type (e.g. **Unit<Length>**). More will be said in § 4.4.

---

4 UCUM: Unified Code for Units Of Measure, see <http://unitsofmeasure.org/>

## 4.4 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 meters units will result in a **ConversionException** being thrown. Rigorous run-time checks are needed because 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 check is not always significant (§ 4.5).

In addition of run-time checks, some limited compile-time checks can be achieved in the Java language using parametrized types. Units can be parametrized with the quantities on which they apply, for example **Unit<Length>** for any units of length. Parametrized types allow applications to put restrictions on the units they can accept.

User can get a parametrized unit by the following means:

- Assignment from one of the predefined constants, for example all constants in the **SI** class.
- All operations returning always a unit of the exact same type than the operand – or, in terms of Java language, all **Unit** methods where the return type is exactly **Unit<Q>**. This include multiplication (**times**) and division (**divide**) by a dimensionless factor.

For example all the following assignments are type safe:

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

However, the assignments below are not type safe, because the result is a type that can not be determined statically by the Java type system. The Java compiler emits a “unchecked cast” warning for such code. Omitting the casts would result in a compiler error.

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

As of Java 5, units checks can't be performed at compile time for such code. However the above code can be rewritten in a slightly safer way as below:

```
Unit<Length> m = Unit.valueOf("m").asType(Length.class);
Unit<Area>    m2 = m.power(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 a **Measure** 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.

For this mechanism to work, users defining new quantity types (on top of the 40+ predefined) have to provide the SI unit of the quantity as a public final static **UNIT** class member of the quantity interface.

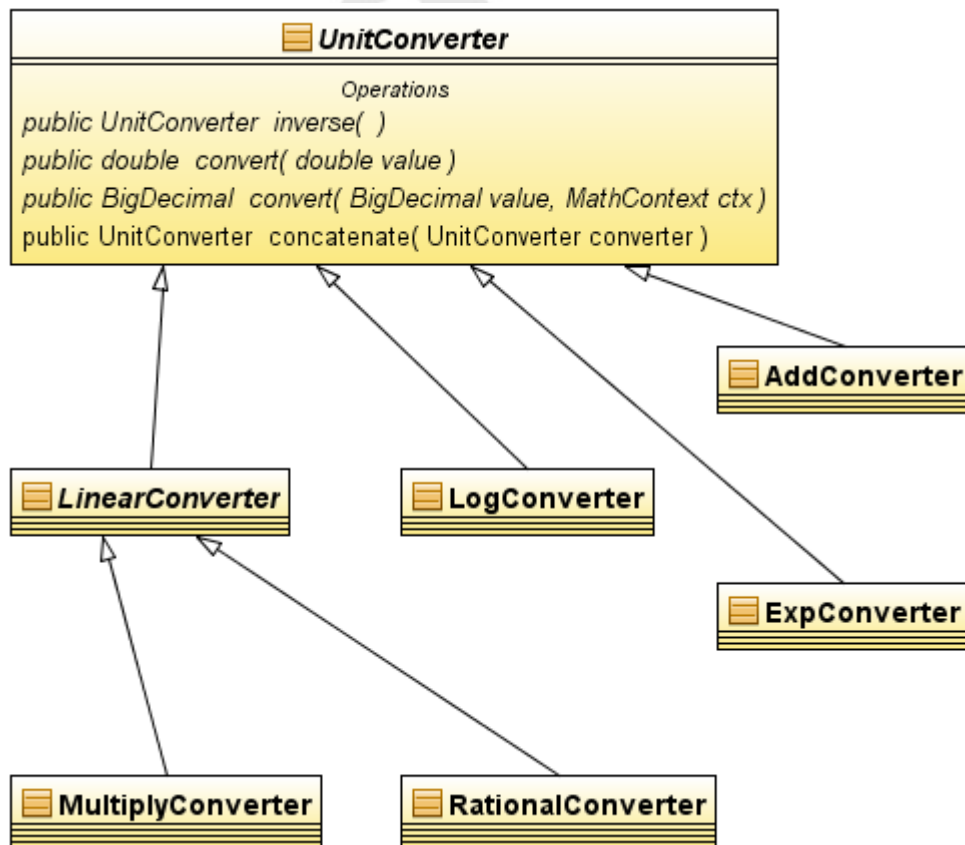
```
public interface Solubility extends Quantity {
    public final static Unit<Solubility> UNIT
        = new ProductUnit<Solubility>(MOLE.divide(CUBIC_METRE));
}
```

### 4.5 Units conversions

Conversions involve two steps: obtain a **UnitConverter** object for a given pair of source and target units, then uses it for converting an arbitrary number of floating point values. The example below converts 4 and 6 meters to 400 and 600 centimeters 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 for a separated **UnitConverter** class as opposed to a **convert(double, Unit)** method straight into the Unit class. 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)**). But 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 kind of unit scale (identity, linear, logarithmic, etc.).



For convenience the **Measure** class provides direct conversion methods such as **Measure.toSI()**, **Measure.to(Unit<Q>)** and **Measure.to(Unit<Q>, MathContext)**.

```
Measure<Velocity> c = Measure.valueOf("299792458 m/s", Velocity.class);
Measure<Velocity> milesPerHour = c.to(MILES_PER_HOUR, DECIMAL128);
System.out.println(milesPerHour);
> 670616629.3843951324266284896206156 [mi_i]/h
```

## 5. Detailed Specification (API)

Package Summary		Page
<a href="#">javax.measure</a>	Provides strongly typed measurements to enforce compile-time check of parameters consistency and avoid interface errors.	17
<a href="#">javax.measure.converter</a>	Provides support for unit conversion.	34
<a href="#">javax.measure.quantity</a>	Provides quantitative properties or attributes of thing such as mass, time, distance, heat, and angular separation.	58
<a href="#">javax.measure.unit</a>	Provides support for programatic unit handling.	112

## Package javax.measure

Provides strongly typed measurements to enforce compile-time check of parameters consistency and avoid interface errors.

See:

[Description](#)

Interface Summary		Page
<a href="#">Measurable</a>	This interface represents the measurable, countable, or comparable property or aspect of a thing.	19

Class Summary		Page
<a href="#">Measure</a>	This class represents the immutable result of a scalar measurement stated in a known unit.	22
<a href="#">MeasureFormat</a>	This class provides the interface for formatting and parsing <a href="#">measurements</a> .	30

## Package javax.measure Description

Provides strongly typed measurements to enforce compile-time check of parameters consistency and avoid interface errors.

Let's take the following example:

```
class Person {
    void setWeight(double weight);
}
```

Should the weight be in pound, kilogram ??

Using measures there is no room for error:

```
class Person {
    void setWeight(Measurable<Mass> weight);
}
```

Not only the interface is cleaner (the weight has to be of mass type); but also there is no confusion on the measurement unit:

```
double weightInKg = weight.doubleValue(KILOGRAM);
double weightInLb = weight.doubleValue(POUND);
```

Measurable work hand-in-hand with units (also parameterized). For example, the following would result in compile-time error:

```
double weightInLiter = weight.doubleValue(LITRE);
// Compile error, Unit<Mass> required.
```

Users may create their own [Measurable](#) implementation:

```
public class Period implements Measurable<Duration> {
    long nanoseconds;
    ...
}
```

```
public class Distance implements Measurable<Length> {
    double meters;
    ...
}

public class Velocity3D implements Measurable<Velocity> {
    double x, y, z; // In meters.
    public double doubleValue(Unit<Velocity> unit) { ... } // Returns vector norm.
    ...
}
```

Users may also combine a definite amount (a number) with a unit and make it a [Measure](#) (and a [Measurable](#) instance).

```
person.setWeight(Measure.valueOf(180.0, POUND)); // Measure<Mass>
timer.setPeriod(Measure.valueOf(20, MILLI(SECOND))); // Measure<Duration>
bottle.setPression(Measure.valueOf("23.4 kg/(m.s2)").to(PASCAL)); // Measure<Pressure>
// or bottle.setPression(Measure.valueOf("23.4 kg/(m.s2)").asType(Pressure.class));
```

Measures may be formatted (or parsed) using either the [standard](#) unit format ([UCUM](#) based) or a locale sensitive format.

```
// Standard format (UCUM). It is used by Measure.valueOf(...) and Measure.toString()
// This format <b>is not</b> locale-sensitive and can be used for unambiguous
// electronic communication of quantities together with their units without loss
// of information.
```

```
Measure<Pressure> pressure = Measure.valueOf("123.45 N/m2").asType(Pressure.class);
// Parse (executes MeasureFormat.getStandard().parse("123.45 N/m2")).
System.out.println(pressure.to(NonSI.BAR));
// Format (executes MeasureFormat.getStandard().format(pressure.to(NonSI.BAR))).
```

```
> 0.0012345 bar
```

```
// Locale format (here French locale). This format is often nicer looking but
// may result in loss of information/precision.
```

```
Measure<Acceleration> acceleration
    = Measure.valueOf(54321, MICRO(METRES_PER_SQUARE_SECOND));
System.out.println(MeasureFormat.getInstance().format(acceleration));
System.out.println(MeasureFormat.getInstance().format(acceleration.toSI()));
```

```
> 54 321 m/s2/1000000
> 0,054 m/s2
```

## Interface Measurable

[javax.measure](#)

### All Superinterfaces:

Comparable<[Measurable](#)<Q>>

### All Known Implementing Classes:

[Measure](#)

```
public interface Measurable
extends Comparable<Measurable<Q>>
```

This interface represents the measurable, countable, or comparable property or aspect of a thing.

Measurable instances are for the most part scalar quantities.

```
class Delay implements Measurable<Duration> {
    private final double seconds; // Implicit internal unit.
    public Delay(double value, Unit<Duration> unit) {
        seconds = unit.getConverterTo(SI.SECOND).convert(value);
    }
    public double doubleValue(Unit<Duration> unit) {
        return SI.SECOND.getConverterTo(unit).convert(seconds);
    }
    ...
}
Thread.wait(new Delay(24, NonSI.HOUR));
// Assuming Thread.wait(Measurable<Duration>) method.
```

Non-scalar quantities are nevertheless allowed as long as an aggregate value makes sense.

```
class Velocity3D implements Measurable<Velocity> {
    private double x, y, z; // Meters per second.
    public double doubleValue(Unit<Velocity> unit) { // Returns the vector norm.
        double metersPerSecond = Math.sqrt(x * x + y * y + z * z);
        return SI.METRES_PER_SECOND.getConverterTo(unit).convert(metersPerSecond);
    }
    ...
}
class ComplexCurrent implements extends Measurable<ElectricCurrent> {
    private Complex amperes;
    public double doubleValue(Unit<ElectricCurrent> unit) { // Returns the magnitude.
        return AMPERE.getConverterTo(unit).convert(amperes.magnitude());
    }
    ...
    public Complex complexValue(Unit<ElectricCurrent> unit) { ... }
}
```

For convenience, measurable instances of any type can be created using the [Measure](#) factory methods.

```
Thread.wait(Measure.valueOf(24, NonSI.HOUR));
```

Method Summary		Page
BigDecimal	<a href="#">decimalValue</a> (Unit<Q> unit, MathContext ctx) Returns the BigDecimal value of this measurable when stated in the specified unit.	21
double	<a href="#">doubleValue</a> (Unit<Q> unit) Returns the double value of this measurable when stated in the specified unit.	21

float	<a href="#">floatValue</a> ( <a href="#">Unit</a> < <a href="#">Q</a> > unit) Returns the <code>float</code> value of this measurable when stated in the specified unit.	20
int	<a href="#">intValue</a> ( <a href="#">Unit</a> < <a href="#">Q</a> > unit) Returns the integral <code>int</code> value of this measurable when stated in the specified unit.	20
long	<a href="#">longValue</a> ( <a href="#">Unit</a> < <a href="#">Q</a> > unit) Returns the integral <code>long</code> value of this measurable when stated in the specified unit.	20

## Method Detail

### intValue

```
int intValue(Unit<Q> unit)
    throws ArithmeticException
```

Returns the integral `int` value of this measurable when stated in the specified unit.

Note: This method differs from the `Number.intValue()` in the sense that an `ArithmeticException` is raised instead of a bit truncation in case of overflow (safety critical).

**Parameters:**

`unit` - the unit in which the returned value is stated.

**Returns:**

the numeric value after conversion to type `int`.

**Throws:**

`ArithmeticException` - if this measurable cannot be represented by a `int` number in the specified unit.

### longValue

```
long longValue(Unit<Q> unit)
    throws ArithmeticException
```

Returns the integral `long` value of this measurable when stated in the specified unit.

Note: This method differs from the `Number.longValue()` in the sense that an `ArithmeticException` is raised instead of a bit truncation in case of overflow (safety critical).

**Parameters:**

`unit` - the unit in which the returned value is stated.

**Returns:**

the numeric value after conversion to type `long`.

**Throws:**

`ArithmeticException` - if this measurable cannot be represented by a `int` number in the specified unit.

### floatValue

```
float floatValue(Unit<Q> unit)
    throws ArithmeticException
```

Returns the `float` value of this measurable when stated in the specified unit. If the measurable has too great of a magnitude to be represented as a `float`, `Float.NEGATIVE_INFINITY` or `Float.POSITIVE_INFINITY` is returned as appropriate.

**Parameters:**

`unit` - the unit in which this returned value is stated.

**Returns:**

the numeric value after conversion to type `float`.

**Throws:**

`ArithmeticException` - if this measurable cannot be represented by a `float` number in the specified unit.

---

## doubleValue

```
double doubleValue (Unit<Q> unit)
    throws ArithmeticException
```

Returns the `double` value of this measurable when stated in the specified unit. If the measurable has too great of a magnitude to be represented as a `double`, `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY` is returned as appropriate.

**Parameters:**

`unit` - the unit in which this returned value is stated.

**Returns:**

the numeric value after conversion to type `double`.

**Throws:**

`ArithmeticException` - if this measurable cannot be represented by a `double` number in the specified unit.

---

## decimalValue

```
BigDecimal decimalValue (Unit<Q> unit,
    MathContext ctx)
    throws ArithmeticException
```

Returns the `BigDecimal` value of this measurable when stated in the specified unit.

**Parameters:**

`unit` - the unit in which the returned value is stated.

`ctx` - the math context being used for conversion.

**Returns:**

the decimal value after conversion.

**Throws:**

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY` or `mathContext.precision == 0` and the quotient has a non-terminating decimal expansion.

# Class Measure

[javax.measure](#)

```
java.lang.Object
└─ javax.measure.Measure
```

## All Implemented Interfaces:

Comparable<[Measurable](#)<Q>>, [Measurable](#)<Q>, Serializable

```
abstract public class Measure
extends Object
implements Measurable<Q>, Serializable
```

This class represents the immutable result of a scalar measurement stated in a known unit.

To avoid any lost of precision, known exact measure (e.g. physical constants) should not be created from `double` constants but from their decimal representation.

```
public static final Measure<Velocity> C
    = Measure.valueOf("299792458 m/s").asType(Velocity.class);
    // Speed of Light (exact).
```

Measures can be converted to different units, the conversion precision is determined by the specified `MathContext`.

```
Measure<Velocity> milesPerHour = C.to(MILES_PER_HOUR, MathContext.DECIMAL128);
    // Use BigDecimal implementation.
System.out.println(milesPerHour);

> 670616629.3843951324266284896206156 [mi_i]/h
```

If no precision is specified `double` precision is assumed.

```
Measure<Velocity> milesPerHour = C.to(MILES_PER_HOUR);
    // Use double implementation (fast).
System.out.println(milesPerHour);

> 670616629.3843951 [mi_i]/h
```

Applications may sub-class [Measure](#) for particular measurements types.

```
// Measurement of type Mass based on <code>double</code> primitive types.
public class Weight extends Measure<Mass> {
    private final double _kilograms; // Internal SI representation.
    private Weight(double kilograms) { _kilograms = kilograms; }
    public static Weight valueOf(double value, Unit<Mass> unit) {
        return new Weight(unit.getConverterTo(SI.KILOGRAM).convert(value));
    }
    public Unit<Mass> getUnit() { return SI.KILOGRAM; }
    public Double getValue() { return _kilograms; }
    ...
}

// Complex numbers measurements.
public class ComplexMeasure<Q> extends Quantity<Q> extends Measure<Q> {
    public Complex getValue() { ... } // Assuming Complex is a Number.
    ...
}

// Specializations of complex numbers measurements.
public class Current extends ComplexMeasure<ElectricCurrent> {...}
public class Tension extends ComplexMeasure<ElectricPotential> {...}
```

All instances of this class shall be immutable.

Constructor Summary		Page
protected	<a href="#">Measure</a> () Default constructor.	24

Method Summary		Page
<a href="#">Measure</a> <T>	<a href="#">asType</a> (Class<T> type) Casts this measure to a parameterized unit of specified nature or throw a <code>ClassCastException</code> if the dimension of the specified quantity and this measure unit's dimension do not match.	28
int	<a href="#">compareTo</a> ( <a href="#">Measurable</a> <Q> that) Compares this measure to the specified measurable quantity.	25
boolean	<a href="#">equals</a> (Object obj) Compares this measure against the specified object for <b>strict</b> equality (same unit and same amount).	25
boolean	<a href="#">equals</a> ( <a href="#">Measurable</a> <Q> that, double epsilon, <a href="#">Unit</a> <Q> epsilonUnit) Compares this measure and the specified measurable to the given accuracy.	26
float	<a href="#">floatValue</a> ( <a href="#">Unit</a> <Q> unit) Returns the <code>float</code> value of this measurable when stated in the specified unit.	27
abstract <a href="#">Unit</a> <Q>	<a href="#">getUnit</a> () Returns the measurement unit.	24
abstract Number	<a href="#">getValue</a> () Returns the measurement numeric value.	24
int	<a href="#">hashCode</a> () Returns the hash code for this measure.	26
int	<a href="#">intValue</a> ( <a href="#">Unit</a> <Q> unit) Returns the integral <code>int</code> value of this measurable when stated in the specified unit.	26
long	<a href="#">longValue</a> ( <a href="#">Unit</a> <Q> unit) Returns the integral <code>long</code> value of this measurable when stated in the specified unit.	27
<a href="#">Measure</a> <Q>	<a href="#">to</a> ( <a href="#">Unit</a> <Q> unit) Returns this measure after conversion to specified unit.	24
<a href="#">Measure</a> <Q>	<a href="#">to</a> ( <a href="#">Unit</a> <Q> unit, <code>MathContext</code> ctx) Returns this measure after conversion to specified unit.	25
<a href="#">Measure</a> <Q>	<a href="#">toSI</a> () Convenient method equivalent to <a href="#">to(this.getUnit().toSI())</a> .	24
String	<a href="#">toString</a> () Returns the <code>String</code> representation of this measure.	26
static <a href="#">Measure</a> <Q>	<a href="#">valueOf</a> (double doubleValue, <a href="#">Unit</a> <Q> unit) Returns the scalar measure for the specified <code>double</code> stated in the specified unit.	29
static <a href="#">Measure</a> <Q>	<a href="#">valueOf</a> (float floatValue, <a href="#">Unit</a> <Q> unit) Returns the scalar measure for the specified <code>float</code> stated in the specified unit.	29
static <a href="#">Measure</a> <Q>	<a href="#">valueOf</a> (int intValue, <a href="#">Unit</a> <Q> unit) Returns the scalar measure for the specified <code>int</code> stated in the specified unit.	28
static <a href="#">Measure</a> <?>	<a href="#">valueOf</a> ( <code>CharSequence</code> csq) Returns the <a href="#">decimal</a> measure of unknown type corresponding to the specified representation.	28
static <a href="#">Measure</a> <Q>	<a href="#">valueOf</a> ( <code>BigDecimal</code> decimalValue, <a href="#">Unit</a> <Q> unit) Returns the scalar measure for the specified <code>BigDecimal</code> stated in the specified unit.	29

<code>static Measure&lt;Q&gt;</code>	<code>valueOf(long longValue, Unit&lt;Q&gt; unit)</code> Returns the scalar measure for the specified <code>long</code> stated in the specified unit.	29
--	--	----

## Constructor Detail

### Measure

protected **Measure**()

Default constructor.

## Method Detail

### getValue

public abstract Number **getValue**()

Returns the measurement numeric value.

**Returns:**  
the measurement value.

### getUnit

public abstract [Unit<Q>](#) **getUnit**()

Returns the measurement unit.

**Returns:**  
the measurement unit.

### toSI

public [Measure<Q>](#) **toSI**()

Convenient method equivalent to [to\(this.getUnit\(\).toSI\(\)\)](#).

**Returns:**  
this measure or a new measure equivalent to this measure but stated in SI units.

**Throws:**  
[ArithmeticException](#) - if the result is inexact and the quotient has a non-terminating decimal expansion.

### to

public [Measure<Q>](#) **to**([Unit<Q>](#) unit)

Returns this measure after conversion to specified unit. The default implementation returns `Measure.valueOf(doubleValue(unit), unit)`. If this measure is already stated in the specified unit, then this measure is returned and no conversion is performed.

**Parameters:**

`unit` - the unit in which the returned measure is stated.

**Returns:**

this measure or a new measure equivalent to this measure but stated in the specified unit.

**Throws:**

`ArithmeticException` - if the result is inexact and the quotient has a non-terminating decimal expansion.

---

## to

```
public Measure<Q> to(Unit<Q> unit,
                    MathContext ctx)
```

Returns this measure after conversion to specified unit. The default implementation returns `Measure.valueOf(decimalValue(unit, ctx), unit)`. If this measure is already stated in the specified unit, then this measure is returned and no conversion is performed.

**Parameters:**

`unit` - the unit in which the returned measure is stated.  
`ctx` - the math context to use for conversion.

**Returns:**

this measure or a new measure equivalent to this measure but stated in the specified unit.

**Throws:**

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY` or `mathContext.precision == 0` and the quotient has a non-terminating decimal expansion.

---

## compareTo

```
public int compareTo(Measurable<Q> that)
```

Compares this measure to the specified measurable quantity. The default implementation compares the `Measurable.doubleValue(Unit)` of both this measure and the specified measurable stated in the same unit (this measure's `unit`).

**Specified by:**

`compareTo` in interface `Comparable`

**Returns:**

a negative integer, zero, or a positive integer as this measure is less than, equal to, or greater than the specified measurable quantity. `Double.compare(this.doubleValue(getUnit()), that.doubleValue(getUnit()))`

---

## equals

```
public boolean equals(Object obj)
```

Compares this measure against the specified object for **strict** equality (same unit and same amount).

Similarly to the `BigDecimal.equals()` method which consider 2.0 and 2.00 as different objects because of different internal scales, measurements such as `Measure.valueOf(3.0, KILOGRAM)` `Measure.valueOf(3, KILOGRAM)` and `Measure.valueOf("3 kg")` might not be considered equals because of possible differences in their implementations.

To compare measures stated using different units or using different amount implementations the `compareTo` or `equals(measurable, epsilon, epsilonUnit)` methods should be used.

**Overrides:**

`equals` in class `Object`

**Parameters:**

`obj` - the object to compare with.

**Returns:**

`this.getUnit.equals(obj.getUnit()) && this.getValue().equals(obj.getValue())`

---

## **equals**

```
public boolean equals(Measurable<Q> that,  
                    double epsilon,  
                    Unit<Q> epsilonUnit)
```

Compares this measure and the specified measurable to the given accuracy. Measurements are considered approximately equals if their absolute differences when stated in the same specified unit is less than the specified epsilon.

**Parameters:**

`that` - the measurable to compare with.

`epsilon` - the absolute error stated in `epsilonUnit`.

`epsilonUnit` - the epsilon unit.

**Returns:**

`abs(this.doubleValue(epsilonUnit) - that.doubleValue(epsilonUnit)) <= epsilon`

---

## **hashCode**

```
public int hashCode()
```

Returns the hash code for this measure.

**Overrides:**

`hashCode` in class `Object`

**Returns:**

the hash code value.

---

## **toString**

```
public final String toString()
```

Returns the `String` representation of this measure. The string produced for a given measure is always the same; it is not affected by locale. This means that it can be used as a canonical string representation for exchanging measure, or as a key for a `Hashtable`, etc. Locale-sensitive measure formatting and parsing is handled by the [MeasureFormat](#) class and its subclasses.

**Overrides:**

`toString` in class `Object`

**Returns:**

`UnitFormat.getInternational().format(this)`

---

## **intValue**

```
public final int intValue(Unit<Q> unit)  
                    throws ArithmeticException
```

**Description copied from interface: [Measurable](#)**

Returns the integral `int` value of this measurable when stated in the specified unit.

Note: This method differs from the `Number.intValue()` in the sense that an `ArithmeticException` is raised instead of a bit truncation in case of overflow (safety critical).

**Specified by:**

[intValue](#) in interface [Measurable](#)

**Parameters:**

`unit` - the unit in which the returned value is stated.

**Returns:**

the numeric value after conversion to type `int`.

**Throws:**

`ArithmeticException` - if this measurable cannot be represented by a `int` number in the specified unit.

---

## longValue

```
public long longValue(Unit<Q> unit)
    throws ArithmeticException
```

**Description copied from interface: [Measurable](#)**

Returns the integral `long` value of this measurable when stated in the specified unit.

Note: This method differs from the `Number.longValue()` in the sense that an `ArithmeticException` is raised instead of a bit truncation in case of overflow (safety critical).

**Specified by:**

[longValue](#) in interface [Measurable](#)

**Parameters:**

`unit` - the unit in which the returned value is stated.

**Returns:**

the numeric value after conversion to type `long`.

**Throws:**

`ArithmeticException` - if this measurable cannot be represented by a `int` number in the specified unit.

---

## floatValue

```
public final float floatValue(Unit<Q> unit)
```

**Description copied from interface: [Measurable](#)**

Returns the `float` value of this measurable when stated in the specified unit. If the measurable has too great of a magnitude to be represented as a `float`, `Float.NEGATIVE_INFINITY` or `Float.POSITIVE_INFINITY` is returned as appropriate.

**Specified by:**

[floatValue](#) in interface [Measurable](#)

**Parameters:**

`unit` - the unit in which this returned value is stated.

**Returns:**

the numeric value after conversion to type `float`.

---

## asType

```
public final Measure<T> asType(Class<T> type)
    throws ClassCastException
```

Casts this measure to a parameterized unit of specified nature or throw a `ClassCastException` if the dimension of the specified quantity and this measure unit's dimension do not match. For example:

```
Measure<Length> length = Measure.valueOf("2 km").asType(Length.class);
```

**Parameters:**

`type` - the quantity class identifying the nature of the measure.

**Returns:**

this measure parameterized with the specified type.

**Throws:**

`ClassCastException` - if the dimension of this unit is different from the specified quantity dimension.

`UnsupportedOperationException` - if the specified quantity class does not have a public static field named "UNIT" holding the SI unit for the quantity.

**See Also:**

[Unit.asType\(Class\)](#)

---

## valueOf

```
public static Measure<?> valueOf(CharSequence csq)
```

Returns the [decimal](#) measure of unknown type corresponding to the specified representation. This method can be used to parse dimensionless quantities.

```
Measure<Dimensionless> proportion = Measure.valueOf("0.234").asType(Dimensionless.class);
```

Note: This method handles only [standard](#) unit format ([UCUM](#) based). Locale-sensitive measure formatting and parsing are handled by the [MeasureFormat](#) class and its subclasses.

**Parameters:**

`csq` - the decimal value and its unit (if any) separated by space(s).

**Returns:**

```
MeasureFormat.getStandard().parse(csq, new ParsePosition(0))
```

---

## valueOf

```
public static Measure<Q> valueOf(int intValue,
    Unit<Q> unit)
```

Returns the scalar measure for the specified `int` stated in the specified unit.

**Parameters:**

`intValue` - the measurement value.

`unit` - the measurement unit.

**Returns:**

the corresponding `int` measure.

---

## valueOf

```
public static Measure<Q> valueOf(long longValue,  
                                Unit<Q> unit)
```

Returns the scalar measure for the specified `long` stated in the specified unit.

**Parameters:**

`longValue` - the measurement value.  
`unit` - the measurement unit.

**Returns:**

the corresponding `long` measure.

---

## valueOf

```
public static Measure<Q> valueOf(float floatValue,  
                                Unit<Q> unit)
```

Returns the scalar measure for the specified `float` stated in the specified unit.

**Parameters:**

`floatValue` - the measurement value.  
`unit` - the measurement unit.

**Returns:**

the corresponding `float` measure.

---

## valueOf

```
public static Measure<Q> valueOf(double doubleValue,  
                                Unit<Q> unit)
```

Returns the scalar measure for the specified `double` stated in the specified unit.

**Parameters:**

`doubleValue` - the measurement value.  
`unit` - the measurement unit.

**Returns:**

the corresponding `double` measure.

---

## valueOf

```
public static Measure<Q> valueOf(BigDecimal decimalValue,  
                                Unit<Q> unit)
```

Returns the scalar measure for the specified `BigDecimal` stated in the specified unit.

**Parameters:**

`decimalValue` - the measurement value.  
`unit` - the measurement unit.

**Returns:**

the corresponding `BigDecimal` measure.

# Class MeasureFormat

[javax.measure](#)

```
java.lang.Object
├─ java.text.Format
│   └─ javax.measure.MeasureFormat
```

**All Implemented Interfaces:**  
Cloneable, Serializable

```
abstract public class MeasureFormat
extends Format
```

This class provides the interface for formatting and parsing [measurements](#).

Instances of this class should be able to format measurements stated in [CompoundUnit](#). See [formatCompound\(...\)](#).

Constructor Summary	Page
<a href="#">MeasureFormat</a> ()	30

Method Summary	Page
StringBuffer <a href="#">format</a> (Object obj, StringBuffer toAppendTo, FieldPosition pos)	32
abstract Appendable <a href="#">format</a> ( <a href="#">Measure</a> <?> measure, Appendable dest) Formats the specified measure into an Appendable.	31
StringBuilder <a href="#">format</a> ( <a href="#">Measure</a> <?> measure, StringBuilder dest) Convenience method equivalent to <a href="#">format(Measure, Appendable)</a> except it does not raise an IOException.	33
protected Appendable <a href="#">formatCompound</a> (double value, <a href="#">CompoundUnit</a> <?> unit, Appendable dest) Formats the specified value using <a href="#">CompoundUnit</a> compound units}.	32
static <a href="#">MeasureFormat</a> <a href="#">getInstance</a> () Returns the measure format for the default locale.	31
static <a href="#">MeasureFormat</a> <a href="#">getInstance</a> ( <a href="#">NumberFormat</a> numberFormat, <a href="#">UnitFormat</a> unitFormat) Returns the measure format using the specified number format and unit format (the number and unit are separated by one space).	31
static <a href="#">MeasureFormat</a> <a href="#">getStandard</a> () Returns the culture invariant format based upon <code>BigDecimal</code> canonical format and the <a href="#">standard</a> unit format.	31
abstract <a href="#">Measure</a> <?> <a href="#">parse</a> (CharSequence csq, ParsePosition cursor) Parses a portion of the specified <code>CharSequence</code> from the specified position to produce an object.	32
<a href="#">Measure</a> <?> <a href="#">parseObject</a> (String source, ParsePosition pos)	33

## Constructor Detail

### MeasureFormat

```
public MeasureFormat()
```

## Method Detail

### `getInstance`

```
public static MeasureFormat getInstance ()
```

Returns the measure format for the default locale. The default format assumes the measure is composed of a decimal number and a [Unit](#) separated by whitespace(s).

**Returns:**

```
MeasureFormat.getInstance(NumberFormat.getInstance(), UnitFormat.getInstance())
```

---

### `getInstance`

```
public static MeasureFormat getInstance(NumberFormat numberFormat,  
                                         UnitFormat unitFormat)
```

Returns the measure format using the specified number format and unit format (the number and unit are separated by one space).

**Parameters:**

`numberFormat` - the number format.

`unitFormat` - the unit format.

**Returns:**

the corresponding format.

---

### `getStandard`

```
public static MeasureFormat getStandard ()
```

Returns the culture invariant format based upon `BigDecimal` canonical format and the [standard](#) unit format. This format **is not** locale-sensitive and can be used for unambiguous electronic communication of quantities together with their units without loss of information. For example: "1.23456789 kg.m/s2"  
returns `Measure.valueOf(new BigDecimal("1.23456789"), Unit.valueOf("kg.m/s2"));`

**Returns:**

the standard measure format.

---

### `format`

```
public abstract Appendable format(Measure<?> measure,  
                                 Appendable dest)  
    throws IOException
```

Formats the specified measure into an `Appendable`.

**Parameters:**

`measure` - the measure to format.

`dest` - the appendable destination.

**Returns:**

the specified `Appendable`.

**Throws:**

`IOException` - if an I/O exception occurs.

---

## parse

```
public abstract Measure<?> parse(CharSequence csq,  
                                ParsePosition cursor)  
    throws IllegalArgumentException
```

Parses a portion of the specified `CharSequence` from the specified position to produce an object. If parsing succeeds, then the index of the `cursor` argument is updated to the index after the last character used.

### Parameters:

`csq` - the `CharSequence` to parse.  
`cursor` - the cursor holding the current parsing index.

### Returns:

the object parsed from the specified character sub-sequence.

### Throws:

`IllegalArgumentException` - if any problem occurs while parsing the specified character sequence (e.g. illegal syntax).

---

## formatCompound

```
protected Appendable formatCompound(double value,  
                                       CompoundUnit<?> unit,  
                                       Appendable dest)  
    throws IOException
```

Formats the specified value using [CompoundUnit](#) compound units}. The default implementation is locale sensitive and does not use space to separate units. For example:

```
Unit<Length> FOOT_INCH = FOOT.compound(INCH);  
Measure<Length> height = Measure.valueOf(1.81, METER);  
System.out.println(height.to(FOOT_INCH));  
  
> 5ft11,26in // French Local  
  
Unit<Angle> DMS = DEGREE_ANGLE.compound(MINUTE_ANGLE).compound(SECOND_ANGLE);  
Measure<Angle> rotation = Measure.valueOf(35.857497, DEGREE_ANGLE);  
System.out.println(rotation.to(DMS));  
  
> 35°51'26,989" // French Local
```

### Parameters:

`value` - the value to format using compound units.  
`unit` - the compound unit.  
`dest` - the appendable destination.

### Returns:

the specified `Appendable`.

### Throws:

`IOException` - if an I/O exception occurs.

---

## format

```
public final StringBuffer format(Object obj,  
                                   StringBuffer toAppendTo,  
                                   FieldPosition pos)
```

### Overrides:

`format` in class `Format`

---

## parseObject

```
public final Measure<?> parseObject(String source,  
                                     ParsePosition pos)
```

### Overrides:

`parseObject` in class `Format`

---

## format

```
public final StringBuilder format(Measure<?> measure,  
                                     StringBuilder dest)
```

Convenience method equivalent to [format\(Measure, Appendable\)](#) except it does not raise an `IOException`.

### Parameters:

`measure` - the measure to format.  
`dest` - the appendable destination.

### Returns:

the specified `StringBuilder`.

## Package **javax.measure.converter**

Provides support for unit conversion.

See:

[Description](#)

Class Summary		Page
<a href="#">AddConverter</a>	This class represents a converter adding a constant offset to numeric values ( <code>double</code> based).	35
<a href="#">ExpConverter</a>	This class represents a exponential converter of limited precision.	40
<a href="#">LinearConverter</a>	This class represents a linear converter.	43
<a href="#">LogConverter</a>	This class represents a logarithmic converter of limited precision.	45
<a href="#">MultiplyConverter</a>	This class represents a converter multiplying numeric values by a constant scaling factor ( <code>double</code> based).	48
<a href="#">RationalConverter</a>	This class represents a converter multiplying numeric values by an exact scaling factor (represented as the quotient of two <code>BigInteger</code> numbers).	51
<a href="#">UnitConverter</a>	This class represents a converter of numeric values.	55

Exception Summary		Page
<a href="#">ConversionException</a>	Signals that a problem of some sort has occurred either when creating a converter between two units or during the conversion itself.	38

## Package **javax.measure.converter** Description

Provides support for unit conversion.

# Class AddConverter

[javax.measure.converter](#)

```
java.lang.Object
├─ javax.measure.converter.UnitConverter
│   └─ javax.measure.converter.AddConverter
```

**All Implemented Interfaces:**  
 Serializable

```
final public class AddConverter
extends UnitConverter
```

This class represents a converter adding a constant offset to numeric values (double based).

Instances of this class are immutable.

Constructor Summary	Page
<a href="#">AddConverter</a> (double offset) Creates an add converter with the specified offset.	35

Method Summary		Page
<a href="#">UnitConverter</a>	<a href="#">concatenate</a> ( <a href="#">UnitConverter</a> converter) Concatenates this converter with another converter.	36
double	<a href="#">convert</a> (double value) Converts a double value.	36
BigDecimal	<a href="#">convert</a> (BigDecimal value, MathContext ctx) Converts a BigDecimal value.	37
boolean	<a href="#">equals</a> (Object obj) Indicates whether this converter is considered to be the the same as the one specified.	37
double	<a href="#">getOffset</a> () Returns the offset value for this add converter.	36
int	<a href="#">hashCode</a> () Returns a hash code value for this converter.	37
<a href="#">AddConverter</a>	<a href="#">inverse</a> () Returns the inverse of this converter.	36
String	<a href="#">toString</a> ()	37

## Constructor Detail

### AddConverter

```
public AddConverter(double offset)
```

Creates an add converter with the specified offset.

**Parameters:**

offset - the offset value.

**Throws:**

IllegalArgumentException - if offset is 0.0 (would result in identity converter).

## Method Detail

### getOffset

```
public double getOffset()
```

Returns the offset value for this add converter.

**Returns:**  
the offset value.

---

### concatenate

```
public UnitConverter concatenate(UnitConverter converter)
```

Concatenates this converter with another converter. The resulting converter is equivalent to first converting by the specified converter, and then converting by this converter.

Note: Implementations must ensure that the [UnitConverter.IDENTITY](#) instance is returned if the resulting converter is an identity converter.

**Overrides:**  
[concatenate](#) in class [UnitConverter](#)

**Parameters:**  
converter - the other converter.

**Returns:**  
the concatenation of this converter with the other converter.

---

### inverse

```
public AddConverter inverse()
```

Returns the inverse of this converter. If  $x$  is a valid value, then  $x == \text{inverse}().\text{convert}(\text{convert}(x))$  to within the accuracy of computer arithmetic.

**Overrides:**  
[inverse](#) in class [UnitConverter](#)

**Returns:**  
the inverse of this converter.

---

### convert

```
public double convert(double value)
```

Converts a double value.

**Overrides:**  
[convert](#) in class [UnitConverter](#)

**Parameters:**  
value - the numeric value to convert.

**Returns:**  
the double value after conversion.

---

## convert

```
public BigDecimal convert(BigDecimal value,  
                           MathContext ctx)  
    throws ArithmeticException
```

Converts a `BigDecimal` value.

**Overrides:**

[convert](#) in class [UnitConverter](#)

**Parameters:**

`value` - the numeric value to convert.  
`ctx` - the math context being used for conversion.

**Returns:**

the decimal value after conversion.

**Throws:**

`ArithmeticException` - if the result is inexact but the rounding mode is `MathContext.UNNECESSARY` or `mathContext.precision == 0` and the quotient has a non-terminating decimal expansion.

---

## toString

```
public final String toString()
```

**Overrides:**

`toString` in class `Object`

---

## equals

```
public boolean equals(Object obj)
```

Indicates whether this converter is considered to be the the same as the one specified.

**Overrides:**

[equals](#) in class [UnitConverter](#)

**Returns:**

`true` if the specified object is a converter considered equals to this converter; `false` otherwise.

---

## hashCode

```
public int hashCode()
```

Returns a hash code value for this converter. Equals object have equal hash codes.

**Overrides:**

[hashCode](#) in class [UnitConverter](#)

**Returns:**

this converter hash code value.

# Class ConversionException

[javax.measure.converter](#)

```

java.lang.Object
├─ java.lang.Throwable
│   └─ java.lang.Exception
│       └─ javax.measure.converter.ConversionException

```

## All Implemented Interfaces:

Serializable

```

public class ConversionException
extends Exception

```

Signals that a problem of some sort has occurred either when creating a converter between two units or during the conversion itself.

Constructor Summary	Page
<a href="#">ConversionException</a> () Constructs a <code>ConversionException</code> with no detail message.	38
<a href="#">ConversionException</a> (String message) Constructs a <code>ConversionException</code> with the specified detail message.	38
<a href="#">ConversionException</a> (String message, Throwable cause) Constructs a <code>ConversionException</code> with the specified detail message and cause.	38
<a href="#">ConversionException</a> (Throwable cause) Constructs a <code>ConversionException</code> with the cause.	39

## Constructor Detail

### ConversionException

```

public ConversionException()

```

Constructs a `ConversionException` with no detail message.

### ConversionException

```

public ConversionException(String message)

```

Constructs a `ConversionException` with the specified detail message.

#### Parameters:

message - the detail message.

### ConversionException

```

public ConversionException(String message,
                          Throwable cause)

```

Constructs a `ConversionException` with the specified detail message and cause.

**Parameters:**

`message` - the detail message.  
`cause` - the cause of the exception.

---

## **ConversionException**

```
public ConversionException(Throwable cause)
```

Constructs a `ConversionException` with the cause.

**Parameters:**

`cause` - the cause of the exception.

# Class **ExpConverter**

[javax.measure.converter](#)

```
java.lang.Object
├─ javax.measure.converter.UnitConverter
│   └─ javax.measure.converter.ExpConverter
```

**All Implemented Interfaces:**

Serializable

```
final public class ExpConverter
extends UnitConverter
```

This class represents a exponential converter of limited precision. Such converter is typically used to create inverse of logarithmic unit.

Instances of this class are immutable.

Constructor Summary	Page
<a href="#">ExpConverter</a> (double base) Creates a logarithmic converter having the specified base.	40

Method Summary		Page
double	<a href="#">convert</a> (double amount) Converts a double value.	42
BigDecimal	<a href="#">convert</a> (BigDecimal value, MathContext ctx) Converts a BigDecimal value.	42
boolean	<a href="#">equals</a> (Object obj) Indicates whether this converter is considered to be the the same as the one specified.	41
double	<a href="#">getBase</a> () Returns the exponential base of this converter.	41
int	<a href="#">hashCode</a> () Returns a hash code value for this converter.	41
<a href="#">UnitConverter</a>	<a href="#">inverse</a> () Returns the inverse of this converter.	41
String	<a href="#">toString</a> ()	41

## Constructor Detail

### ExpConverter

```
public ExpConverter (double base)
```

Creates a logarithmic converter having the specified base.

**Parameters:**

base - the logarithmic base (e.g. `Math.E` for the Natural Logarithm).

## Method Detail

### getBase

```
public double getBase()
```

Returns the exponential base of this converter.

**Returns:**

the exponential base (e.g. `Math.E` for the Natural Exponential).

---

### inverse

```
public UnitConverter inverse()
```

Returns the inverse of this converter. If `x` is a valid value, then `x == inverse().convert(convert(x))` to within the accuracy of computer arithmetic.

**Overrides:**

[inverse](#) in class [UnitConverter](#)

**Returns:**

the inverse of this converter.

---

### toString

```
public final String toString()
```

**Overrides:**

`toString` in class `Object`

---

### equals

```
public boolean equals(Object obj)
```

Indicates whether this converter is considered to be the the same as the one specified.

**Overrides:**

[equals](#) in class [UnitConverter](#)

**Returns:**

`true` if the specified object is a converter considered equals to this converter; `false` otherwise.

---

### hashCode

```
public int hashCode()
```

Returns a hash code value for this converter. Equals object have equal hash codes.

**Overrides:**

[hashCode](#) in class [UnitConverter](#)

**Returns:**

this converter hash code value.

## convert

```
public double convert(double amount)
```

Converts a double value.

**Overrides:**

[convert](#) in class [UnitConverter](#)

**Returns:**

the double value after conversion.

---

## convert

```
public BigDecimal convert(BigDecimal value,  
                           MathContext ctx)  
    throws ArithmeticException
```

Converts a BigDecimal value.

**Overrides:**

[convert](#) in class [UnitConverter](#)

**Parameters:**

value - the numeric value to convert.

ctx - the math context being used for conversion.

**Returns:**

the decimal value after conversion.

**Throws:**

`ArithmeticException` - if the result is inexact but the rounding mode is `MathContext.UNNECESSARY` or `mathContext.precision == 0` and the quotient has a non-terminating decimal expansion.

# Class LinearConverter

[javax.measure.converter](#)

```
java.lang.Object
├─ javax.measure.converter.UnitConverter
│   └─ javax.measure.converter.LinearConverter
```

**All Implemented Interfaces:**

Serializable

**Direct Known Subclasses:**

[MultiplyConverter](#), [RationalConverter](#)

```
abstract public class LinearConverter
extends UnitConverter
```

This class represents a linear converter. A converter is linear if  $convert(u + v) == convert(u) + convert(v)$  and  $convert(r * u) == r * convert(u)$ . For linear converters the following property always hold:

```
y1 = c1.convert(x1);
y2 = c2.convert(x2);
// then y1*y2 == c1.concatenate(c2).convert(x1*x2)
```

[Concatenation](#) of linear converters always result into a linear converter.

Constructor Summary		Page
<a href="#">LinearConverter</a>	()	43

Method Summary		Page
<a href="#">UnitConverter</a>	<b>concatenate</b> ( <a href="#">UnitConverter</a> converter)	43
	Concatenates this converter with another converter.	
abstract <a href="#">LinearConverter</a>	<b>inverse</b> ()	44
	Returns the inverse of this converter.	

## Constructor Detail

### LinearConverter

```
public LinearConverter()
```

## Method Detail

### concatenate

```
public UnitConverter concatenate(UnitConverter converter)
```

Concatenates this converter with another converter. The resulting converter is equivalent to first converting by the specified converter, and then converting by this converter.

Note: Implementations must ensure that the [UnitConverter.IDENTITY](#) instance is returned if the resulting converter is an identity converter.

**Overrides:**

[concatenate](#) in class [UnitConverter](#)

**Parameters:**

`converter` - the other converter.

**Returns:**

the concatenation of this converter with the other converter.

---

**inverse**

```
public abstract LinearConverter inverse()
```

Returns the inverse of this converter. If `x` is a valid value, then `x == inverse().convert(convert(x))` to within the accuracy of computer arithmetic.

**Overrides:**

[inverse](#) in class [UnitConverter](#)

**Returns:**

the inverse of this converter.

# Class LogConverter

[javax.measure.converter](#)

```

java.lang.Object
├─ javax.measure.converter.UnitConverter
│   └─ javax.measure.converter.LogConverter

```

## All Implemented Interfaces:

Serializable

```

final public class LogConverter
extends UnitConverter

```

This class represents a logarithmic converter of limited precision. Such converter is typically used to create logarithmic unit. For example:

```
Unit<Dimensionless> BEL = Unit.ONE.transform(new LogConverter(10).inverse());
```

Instances of this class are immutable.

Constructor Summary	Page
<a href="#">LogConverter</a> (double base) Creates a logarithmic converter having the specified base.	45

Method Summary	Page
double <a href="#">convert</a> (double amount) Converts a double value.	47
BigDecimal <a href="#">convert</a> (BigDecimal value, MathContext ctx) Converts a BigDecimal value.	47
boolean <a href="#">equals</a> (Object obj) Indicates whether this converter is considered to be the the same as the one specified.	46
double <a href="#">getBase</a> () Returns the logarithmic base of this converter.	46
int <a href="#">hashCode</a> () Returns a hash code value for this converter.	46
<a href="#">UnitConverter</a> <a href="#">inverse</a> () Returns the inverse of this converter.	46
String <a href="#">toString</a> ()	46

## Constructor Detail

### LogConverter

```
public LogConverter (double base)
```

Creates a logarithmic converter having the specified base.

#### Parameters:

base - the logarithmic base (e.g. `Math.E` for the Natural Logarithm).

## Method Detail

### getBase

```
public double getBase()
```

Returns the logarithmic base of this converter.

**Returns:**

the logarithmic base (e.g. `Math.E` for the Natural Logarithm).

---

### inverse

```
public UnitConverter inverse()
```

Returns the inverse of this converter. If `x` is a valid value, then `x == inverse().convert(convert(x))` to within the accuracy of computer arithmetic.

**Overrides:**

[inverse](#) in class [UnitConverter](#)

**Returns:**

the inverse of this converter.

---

### toString

```
public final String toString()
```

**Overrides:**

`toString` in class `Object`

---

### equals

```
public boolean equals(Object obj)
```

Indicates whether this converter is considered to be the the same as the one specified.

**Overrides:**

[equals](#) in class [UnitConverter](#)

**Returns:**

`true` if the specified object is a converter considered equals to this converter; `false` otherwise.

---

### hashCode

```
public int hashCode()
```

Returns a hash code value for this converter. Equals object have equal hash codes.

**Overrides:**

[hashCode](#) in class [UnitConverter](#)

**Returns:**

this converter hash code value.

## convert

```
public double convert(double amount)
```

Converts a double value.

**Overrides:**

[convert](#) in class [UnitConverter](#)

**Returns:**

the double value after conversion.

---

## convert

```
public BigDecimal convert(BigDecimal value,  
                           MathContext ctx)  
    throws ArithmeticException
```

Converts a BigDecimal value.

**Overrides:**

[convert](#) in class [UnitConverter](#)

**Parameters:**

value - the numeric value to convert.

ctx - the math context being used for conversion.

**Returns:**

the decimal value after conversion.

**Throws:**

`ArithmeticException` - if the result is inexact but the rounding mode is `MathContext.UNNECESSARY` or `mathContext.precision == 0` and the quotient has a non-terminating decimal expansion.

# Class MultiplyConverter

[javax.measure.converter](#)

```
java.lang.Object
├─ javax.measure.converter.UnitConverter
│   └─ javax.measure.converter.LinearConverter
│       └─ javax.measure.converter.MultiplyConverter
```

**All Implemented Interfaces:**

Serializable

```
final public class MultiplyConverter
extends LinearConverter
```

This class represents a converter multiplying numeric values by a constant scaling factor (double based).

Instances of this class are immutable.

Constructor Summary		Page
<a href="#">MultiplyConverter</a> (double factor)	Creates a multiply converter with the specified scale factor.	48

Method Summary		Page
<a href="#">UnitConverter</a>	<a href="#">concatenate</a> ( <a href="#">UnitConverter</a> converter) Concatenates this converter with another converter.	49
double	<a href="#">convert</a> (double value) Converts a double value.	49
BigDecimal	<a href="#">convert</a> (BigDecimal value, MathContext ctx) Converts a BigDecimal value.	50
boolean	<a href="#">equals</a> (Object obj) Indicates whether this converter is considered to be the the same as the one specified.	50
double	<a href="#">getFactor</a> () Returns the scale factor of this converter.	49
int	<a href="#">hashCode</a> () Returns a hash code value for this converter.	50
<a href="#">MultiplyConverter</a>	<a href="#">inverse</a> () Returns the inverse of this converter.	49
String	<a href="#">toString</a> ()	50

## Constructor Detail

### MultiplyConverter

```
public MultiplyConverter(double factor)
```

Creates a multiply converter with the specified scale factor.

**Parameters:**

factor - the scaling factor.

**Throws:**

`IllegalArgumentException` - if coefficient is 1.0 (would result in identity converter)

## Method Detail

### getFactor

```
public double getFactor()
```

Returns the scale factor of this converter.

**Returns:**

the scale factor.

---

### concatenate

```
public UnitConverter concatenate(UnitConverter converter)
```

Concatenates this converter with another converter. The resulting converter is equivalent to first converting by the specified converter, and then converting by this converter.

Note: Implementations must ensure that the [UnitConverter.IDENTITY](#) instance is returned if the resulting converter is an identity converter.

**Overrides:**

[concatenate](#) in class [LinearConverter](#)

**Parameters:**

`converter` - the other converter.

**Returns:**

the concatenation of this converter with the other converter.

---

### inverse

```
public MultiplyConverter inverse()
```

Returns the inverse of this converter. If `x` is a valid value, then `x == inverse().convert(convert(x))` to within the accuracy of computer arithmetic.

**Overrides:**

[inverse](#) in class [LinearConverter](#)

**Returns:**

the inverse of this converter.

---

### convert

```
public double convert(double value)
```

Converts a `double` value.

**Overrides:**

[convert](#) in class [UnitConverter](#)

**Parameters:**

`value` - the numeric value to convert.

**Returns:**  
the `double` value after conversion.

---

## convert

```
public BigDecimal convert(BigDecimal value,  
                           MathContext ctx)  
    throws ArithmeticException
```

Converts a `BigDecimal` value.

**Overrides:**  
[convert](#) in class [UnitConverter](#)

**Parameters:**  
`value` - the numeric value to convert.  
`ctx` - the math context being used for conversion.

**Returns:**  
the decimal value after conversion.

**Throws:**  
`ArithmeticException` - if the result is inexact but the rounding mode is `MathContext.UNNECESSARY` or `mathContext.precision == 0` and the quotient has a non-terminating decimal expansion.

---

## toString

```
public final String toString()
```

**Overrides:**  
`toString` in class `Object`

---

## equals

```
public boolean equals(Object obj)
```

Indicates whether this converter is considered to be the the same as the one specified.

**Overrides:**  
[equals](#) in class [UnitConverter](#)

**Returns:**  
`true` if the specified object is a converter considered equals to this converter;`false` otherwise.

---

## hashCode

```
public int hashCode()
```

Returns a hash code value for this converter. Equals object have equal hash codes.

**Overrides:**  
[hashCode](#) in class [UnitConverter](#)

**Returns:**  
this converter hash code value.

# Class RationalConverter

[javax.measure.converter](#)

```

java.lang.Object
├── javax.measure.converter.UnitConverter
│   └── javax.measure.converter.LinearConverter
│       └── javax.measure.converter.RationalConverter
    
```

**All Implemented Interfaces:**

Serializable

```

final public class RationalConverter
extends LinearConverter
    
```

This class represents a converter multiplying numeric values by an exact scaling factor (represented as the quotient of two `BigInteger` numbers).

Instances of this class are immutable.

Constructor Summary	Page
<a href="#">RationalConverter</a> ( <code>BigInteger</code> dividend, <code>BigInteger</code> divisor) Creates a rational converter with the specified dividend and divisor.	51

Method Summary	Page
<a href="#">UnitConverter</a> <a href="#">concatenate</a> ( <a href="#">UnitConverter</a> converter) Concatenates this converter with another converter.	53
<code>double</code> <a href="#">convert</a> ( <code>double</code> value) Converts a <code>double</code> value.	52
<code>BigDecimal</code> <a href="#">convert</a> ( <code>BigDecimal</code> value, <code>MathContext</code> ctx) Converts a <code>BigDecimal</code> value.	52
<code>boolean</code> <a href="#">equals</a> ( <code>Object</code> obj) Indicates whether this converter is considered to be the the same as the one specified.	53
<code>BigInteger</code> <a href="#">getDividend</a> () Returns the integer dividend for this rational converter.	52
<code>BigInteger</code> <a href="#">getDivisor</a> () Returns the integer (positive) divisor for this rational converter.	52
<code>int</code> <a href="#">hashCode</a> () Returns a hash code value for this converter.	54
<a href="#">RationalConverter</a> <a href="#">inverse</a> () Returns the inverse of this converter.	53
<code>String</code> <a href="#">toString</a> ()	53

## Constructor Detail

### RationalConverter

```

public RationalConverter(BigInteger dividend,
                        BigInteger divisor)
    
```

Creates a rational converter with the specified dividend and divisor.

**Parameters:**

dividend - the dividend.  
divisor - the positive divisor.

**Throws:**

`IllegalArgumentException` - if divisor <= 0,  
if dividend == divisor

## Method Detail

### getDividend

```
public BigInteger getDividend()
```

Returns the integer dividend for this rational converter.

**Returns:**

this converter dividend.

---

### getDivisor

```
public BigInteger getDivisor()
```

Returns the integer (positive) divisor for this rational converter.

**Returns:**

this converter divisor.

---

### convert

```
public double convert(double value)
```

Converts a double value.

**Overrides:**

[convert](#) in class [UnitConverter](#)

**Parameters:**

value - the numeric value to convert.

**Returns:**

the double value after conversion.

---

### convert

```
public BigDecimal convert(BigDecimal value,  
                           MathContext ctx)  
    throws ArithmeticException
```

Converts a `BigDecimal` value.

**Overrides:**

[convert](#) in class [UnitConverter](#)

**Parameters:**

value - the numeric value to convert.  
ctx - the math context being used for conversion.

**Returns:**

the decimal value after conversion.

**Throws:**

`ArithmeticException` - if the result is inexact but the rounding mode is `MathContext.UNNECESSARY` or `mathContext.precision == 0` and the quotient has a non-terminating decimal expansion.

---

## concatenate

```
public UnitConverter concatenate(UnitConverter converter)
```

Concatenates this converter with another converter. The resulting converter is equivalent to first converting by the specified converter, and then converting by this converter.

Note: Implementations must ensure that the [UnitConverter.IDENTITY](#) instance is returned if the resulting converter is an identity converter.

**Overrides:**

[concatenate](#) in class [LinearConverter](#)

**Parameters:**

`converter` - the other converter.

**Returns:**

the concatenation of this converter with the other converter.

---

## inverse

```
public RationalConverter inverse()
```

Returns the inverse of this converter. If `x` is a valid value, then `x == inverse().convert(convert(x))` to within the accuracy of computer arithmetic.

**Overrides:**

[inverse](#) in class [LinearConverter](#)

**Returns:**

the inverse of this converter.

---

## toString

```
public final String toString()
```

**Overrides:**

`toString` in class `Object`

---

## equals

```
public boolean equals(Object obj)
```

Indicates whether this converter is considered to be the the same as the one specified.

**Overrides:**

[equals](#) in class [UnitConverter](#)

**Returns:**

`true` if the specified object is a converter considered equals to this converter; `false` otherwise.

## hashCode

```
public int hashCode()
```

Returns a hash code value for this converter. Equals object have equal hash codes.

**Overrides:**

[hashCode](#) in class [UnitConverter](#)

**Returns:**

this converter hash code value.

# Class `UnitConverter`

[javax.measure.converter](#)

```
java.lang.Object
└─ javax.measure.converter.UnitConverter
```

## All Implemented Interfaces:

`Serializable`

## Direct Known Subclasses:

[AddConverter](#), [ExpConverter](#), [LinearConverter](#), [LogConverter](#)

```
abstract public class UnitConverter
extends Object
implements Serializable
```

This class represents a converter of numeric values.

It is not required for sub-classes to be immutable (e.g. currency converter).

Sub-classes must ensure unicity of the [identity](#) converter. In other words, if the result of an operation is equivalent to the identity converter, then the unique [IDENTITY](#) instance should be returned.

Field Summary		Page
<code>static UnitConverter</code>	<a href="#">IDENTITY</a> Holds the identity converter (unique).	55

Constructor Summary		Page
<code>protected UnitConverter ()</code>	Default constructor.	56

Method Summary		Page
<code>UnitConverter</code>	<a href="#">concatenate</a> ( <code>UnitConverter</code> converter) Concatenates this converter with another converter.	57
<code>abstract double</code>	<a href="#">convert</a> (double value) Converts a double value.	56
<code>abstract BigDecimal</code>	<a href="#">convert</a> (BigDecimal value, MathContext ctx) Converts a BigDecimal value.	56
<code>abstract boolean</code>	<a href="#">equals</a> (Object cvtr) Indicates whether this converter is considered to be the the same as the one specified.	57
<code>abstract int</code>	<a href="#">hashCode</a> () Returns a hash code value for this converter.	57
<code>abstract UnitConverter</code>	<a href="#">inverse</a> () Returns the inverse of this converter.	56

## Field Detail

### IDENTITY

```
public static final UnitConverter IDENTITY
```

Holds the identity converter (unique). This converter does nothing (`ONE.convert(x) == x`). This instance is unique. (

## Constructor Detail

### UnitConverter

```
protected UnitConverter()
```

Default constructor.

## Method Detail

### inverse

```
public abstract UnitConverter inverse()
```

Returns the inverse of this converter. If `x` is a valid value, then `x == inverse().convert(convert(x))` to within the accuracy of computer arithmetic.

**Returns:**  
the inverse of this converter.

---

### convert

```
public abstract double convert(double value)
```

Converts a `double` value.

**Parameters:**  
`value` - the numeric value to convert.

**Returns:**  
the `double` value after conversion.

---

### convert

```
public abstract BigDecimal convert(BigDecimal value,  
                                     MathContext ctx)  
    throws ArithmeticException
```

Converts a `BigDecimal` value.

**Parameters:**  
`value` - the numeric value to convert.  
`ctx` - the math context being used for conversion.

**Returns:**  
the decimal value after conversion.

**Throws:**  
`ArithmeticException` - if the result is inexact but the rounding mode is `MathContext.UNNECESSARY` or `mathContext.precision == 0` and the quotient has a non-terminating decimal expansion.

## equals

```
public abstract boolean equals(Object cvtr)
```

Indicates whether this converter is considered to be the the same as the one specified.

**Overrides:**

`equals` in class `Object`

**Parameters:**

`cvtr` - the converter with which to compare.

**Returns:**

`true` if the specified object is a converter considered equals to this converter;`false` otherwise.

---

## hashCode

```
public abstract int hashCode()
```

Returns a hash code value for this converter. Equals object have equal hash codes.

**Overrides:**

`hashCode` in class `Object`

**Returns:**

this converter hash code value.

**See Also:**

[equals\(\)](#)

---

## concatenate

```
public UnitConverter concatenate(UnitConverter converter)
```

Concatenates this converter with another converter. The resulting converter is equivalent to first converting by the specified converter, and then converting by this converter.

Note: Implementations must ensure that the [IDENTITY](#) instance is returned if the resulting converter is an identity converter.

**Parameters:**

`converter` - the other converter.

**Returns:**

the concatenation of this converter with the other converter.

## Package `javax.measure.quantity`

Provides quantitative properties or attributes of thing such as mass, time, distance, heat, and angular separation.

See:

[Description](#)

Interface Summary		Page
<a href="#">Acceleration</a>	This interface represents the rate of change of velocity with respect to time.	60
<a href="#">Action</a>	This interface represents an energy multiplied by a duration (quantity associated to the <a href="#">Planck Constant</a> ).	61
<a href="#">AmountOfSubstance</a>	This interface represents the number of elementary entities (molecules, for example) of a substance.	62
<a href="#">Angle</a>	This interface represents the figure formed by two lines diverging from a common point.	63
<a href="#">AngularAcceleration</a>	This interface represents the rate of change of angular velocity with respect to time.	64
<a href="#">AngularVelocity</a>	This interface represents the rate of change of angular displacement with respect to time.	65
<a href="#">Area</a>	This interface represents the extent of a planar region or of the surface of a solid measured in square units.	66
<a href="#">CatalyticActivity</a>	This interface represents a catalytic activity.	67
<a href="#">DataAmount</a>	This interface represents a measure of data amount.	68
<a href="#">DataRate</a>	This interface represents the speed of data-transmission.	69
<a href="#">Dimensionless</a>	This interface represents a dimensionless quantity.	70
<a href="#">Duration</a>	This interface represents a period of existence or persistence.	71
<a href="#">DynamicViscosity</a>	This interface represents the dynamic viscosity.	72
<a href="#">ElectricCapacitance</a>	This interface represents an electric capacitance.	73
<a href="#">ElectricCharge</a>	This interface represents an electric charge.	74
<a href="#">ElectricConductance</a>	This interface represents an electric conductance.	75
<a href="#">ElectricCurrent</a>	This interface represents the amount of electric charge flowing past a specified circuit point per unit time.	76
<a href="#">ElectricInductance</a>	This interface represents an electric inductance.	77
<a href="#">ElectricPermittivity</a>	This interface represents how an electric field affects, and is affected by a dielectric medium.	78
<a href="#">ElectricPotential</a>	This interface represents an electric potential or electromotive force.	79
<a href="#">ElectricResistance</a>	This interface represents an electric resistance.	80
<a href="#">Energy</a>	This interface represents the capacity of a physical system to do work.	81
<a href="#">Force</a>	This interface represents a quantity that tends to produce an acceleration of a body in the direction of its application.	82
<a href="#">Frequency</a>	This interface represents the number of times a specified phenomenon occurs within a specified interval.	83
<a href="#">Illuminance</a>	This interface represents an illuminance.	84
<a href="#">IonizingRadiation</a>	This interface represents the quantity of subatomic particles or electromagnetic waves that are energetic enough to detach electrons from atoms or molecules, ionizing them.	85
<a href="#">KinematicViscosity</a>	This interface represents the diffusion of momentum.	86

<a href="#"><u>Length</u></a>	This interface represents the extent of something along its greatest dimension or the extent of space between two objects or places.	87
<a href="#"><u>Luminance</u></a>	This interface represents the luminous intensity per unit area of light traveling in a given direction.	88
<a href="#"><u>LuminousFlux</u></a>	This interface represents a luminous flux.	89
<a href="#"><u>LuminousIntensity</u></a>	This interface represents the luminous flux density per solid angle as measured in a given direction relative to the emitting source.	90
<a href="#"><u>MagneticFieldStrength</u></a>	This interface represents a magnetic field strength.	91
<a href="#"><u>MagneticFlux</u></a>	This interface represents a magnetic flux.	92
<a href="#"><u>MagneticFluxDensity</u></a>	This interface represents a magnetic flux density.	93
<a href="#"><u>MagneticPermeability</u></a>	This interface represents the degree of magnetization of a material that responds linearly to an applied magnetic field.	94
<a href="#"><u>MagnetomotiveForce</u></a>	This interface represents a force that produces magnetic flux.	95
<a href="#"><u>Mass</u></a>	This interface represents the measure of the quantity of matter that a body or an object contains.	96
<a href="#"><u>MassFlowRate</u></a>	This interface represents the movement of mass per time.	97
<a href="#"><u>Power</u></a>	This interface represents the rate at which work is done.	98
<a href="#"><u>Pressure</u></a>	This interface represents a force applied uniformly over a surface.	99
<a href="#"><u>Quantity</u></a>	This interface represents any type of quantitative properties or attributes of thing.	100
<a href="#"><u>RadiationDoseAbsorbed</u></a>	This interface represents the amount of energy deposited per unit of mass.	101
<a href="#"><u>RadiationDoseEffective</u></a>	This interface represents the effective (or "equivalent") dose of radiation received by a human or some other living organism.	102
<a href="#"><u>RadioactiveActivity</u></a>	This interface represents a radioactive activity.	103
<a href="#"><u>SolidAngle</u></a>	This interface represents the angle formed by three or more planes intersecting at a common point.	104
<a href="#"><u>Temperature</u></a>	This class represents the degree of hotness or coldness of a body or an environment.	105
<a href="#"><u>Torque</u></a>	This interface represents the moment of a force.	106
<a href="#"><u>Velocity</u></a>	This interface represents a distance traveled divided by the time of travel.	107
<a href="#"><u>Volume</u></a>	This interface represents the amount of space occupied by a three-dimensional object or region of space, expressed in cubic units.	108
<a href="#"><u>VolumetricDensity</u></a>	This interface represents a mass per unit volume of a substance under specified conditions of pressure and temperature.	109
<a href="#"><u>VolumetricFlowRate</u></a>	This interface represents the volume of fluid passing a point in a system per unit of time.	110
<a href="#"><u>Wavenumber</u></a>	This interface represents a wave property inversely related to wavelength.	111

## Package `javax.measure.quantity` Description

Provides quantitative properties or attributes of thing such as mass, time, distance, heat, and angular separation.

Each quantity sub-interface holds a static `UNIT` field holding the standard SI unit for the quantity.

## Interface Acceleration

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

```
public interface Acceleration
extends Quantity
```

This interface represents the rate of change of velocity with respect to time. The system unit for this quantity is "m/s<sup>2</sup>" (metre per square second).

Field Summary		Page
<a href="#">Unit&lt;Acceleration&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	60

### Field Detail

#### UNIT

```
public static final Unit<Acceleration> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Action

[javax.measure.quantity](#)

### All Superinterfaces:

[Quantity](#)

---

```
public interface Action
extends Quantity
```

This interface represents an energy multiplied by a duration (quantity associated to the [Planck Constant](#)). The system unit for this quantity is "J.s" (joules second).

### See Also:

[Wikipedia's Action](#)

---

Field Summary		Page
<a href="#">Unit&lt;Action&gt;</a> <b>UNIT</b>	Holds the SI unit (Système International d'Unités) for this quantity.	61

## Field Detail

### UNIT

```
public static final Unit<Action> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface AmountOfSubstance

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface AmountOfSubstance
extends Quantity
```

This interface represents the number of elementary entities (molecules, for example) of a substance. The system unit for this quantity is "mol" (mole).

---

Field Summary		Page
<a href="#">Unit&lt;AmountOfSubstance&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	62

### Field Detail

#### UNIT

```
public static final Unit<AmountOfSubstance> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Angle

[javax.measure.quantity](#)

All Superinterfaces:

[Dimensionless](#), [Quantity](#)

---

```
public interface Angle
extends Dimensionless
```

This interface represents the figure formed by two lines diverging from a common point. The system unit for this quantity is "rad" (radian). This quantity is dimensionless.

---

Field Summary		Page
<a href="#">Unit&lt;Angle&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	63

### Field Detail

#### UNIT

```
public static final Unit<Angle> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface AngularAcceleration

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface AngularAcceleration  
extends Quantity
```

This interface represents the rate of change of angular velocity with respect to time. The system unit for this quantity is "rad/s<sup>2</sup>" (radian per square second).

---

Field Summary		Page
<a href="#">Unit&lt;AngularAcceleration&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	64

### Field Detail

#### UNIT

```
public static final Unit<AngularAcceleration> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface AngularVelocity

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface AngularVelocity
extends Quantity
```

This interface represents the rate of change of angular displacement with respect to time. The system unit for this quantity is "rad/s" (radian per second).

---

Field Summary		Page
<a href="#">Unit&lt;AngularVelocity&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	65

### Field Detail

#### UNIT

```
public static final Unit<AngularVelocity> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Area

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Area
extends Quantity
```

This interface represents the extent of a planar region or of the surface of a solid measured in square units. The system unit for this quantity is "m2" (square metre).

---

Field Summary		Page
<a href="#">Unit&lt;Area&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	66

## Field Detail

### UNIT

```
public static final Unit<Area> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface CatalyticActivity

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface CatalyticActivity
extends Quantity
```

This interface represents a catalytic activity. The system unit for this quantity is "kat" (katal).

---

Field Summary		Page
<a href="#">Unit&lt;CatalyticActivity&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	67

### Field Detail

#### UNIT

```
public static final Unit<CatalyticActivity> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface DataAmount

[javax.measure.quantity](#)

All Superinterfaces:

[Dimensionless](#), [Quantity](#)

---

```
public interface DataAmount
extends Dimensionless
```

This interface represents a measure of data amount. The system unit for this quantity is "bit". This quantity is dimensionless.

---

Field Summary		Page
<a href="#">Unit&lt;DataAmount&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	68

### Field Detail

#### UNIT

```
public static final Unit<DataAmount> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface DataRate

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface DataRate  
extends Quantity
```

This interface represents the speed of data-transmission. The system unit for this quantity is "bit/s" (bit per second).

---

Field Summary		Page
<a href="#">Unit&lt;DataRate&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	69

## Field Detail

### UNIT

```
public static final Unit<DataRate> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Dimensionless

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

All Known Subinterfaces:

[Angle](#), [DataAmount](#), [SolidAngle](#)

```
public interface Dimensionless
extends Quantity
```

This interface represents a dimensionless quantity.

Field Summary		Page
<a href="#">Unit&lt;Dimensionless&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	70

### Field Detail

#### UNIT

```
public static final Unit<Dimensionless> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface **Duration**

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Duration
extends Quantity
```

This interface represents a period of existence or persistence. The system unit for this quantity is "s" (second).

---

Field Summary		Page
<a href="#">Unit&lt;Duration&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	71

### Field Detail

#### UNIT

```
public static final Unit<Duration> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface **DynamicViscosity**

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface DynamicViscosity  
extends Quantity
```

This interface represents the dynamic viscosity. The system unit for this quantity is "Pa.s" (Pascal-Second).

See Also:

[Wikipedia: Viscosity](#)

---

Field Summary		Page
<a href="#">Unit&lt;DynamicViscosity&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	72

### Field Detail

#### UNIT

```
public static final Unit<DynamicViscosity> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface **ElectricCapacitance**

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface ElectricCapacitance  
extends Quantity
```

This interface represents an electric capacitance. The system unit for this quantity is "F" (Farad).

---

Field Summary		Page
<a href="#">Unit&lt;ElectricCapacitance&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	73

### Field Detail

#### UNIT

```
public static final Unit<ElectricCapacitance> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface **ElectricCharge**

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface ElectricCharge
extends Quantity
```

This interface represents an electric charge. The system unit for this quantity is "C" (Coulomb).

---

Field Summary		Page
<a href="#">Unit&lt;ElectricCharge&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	74

### Field Detail

#### UNIT

```
public static final Unit<ElectricCharge> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface **ElectricConductance**

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface ElectricConductance  
extends Quantity
```

This interface represents an electric conductance. The system unit for this quantity "S" (Siemens).

---

Field Summary		Page
<a href="#">Unit&lt;ElectricConductance&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	75

### Field Detail

#### UNIT

```
public static final Unit<ElectricConductance> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface **ElectricCurrent**

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface ElectricCurrent  
extends Quantity
```

This interface represents the amount of electric charge flowing past a specified circuit point per unit time. The system unit for this quantity is "A" (Ampere).

---

Field Summary		Page
<a href="#">Unit&lt;ElectricCurrent&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	76

### Field Detail

#### UNIT

```
public static final Unit<ElectricCurrent> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface **ElectricInductance**

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface ElectricInductance  
extends Quantity
```

This interface represents an electric inductance. The system unit for this quantity is "H" (Henry).

---

Field Summary		Page
<a href="#">Unit&lt;ElectricInductance&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	77

### Field Detail

#### UNIT

```
public static final Unit<ElectricInductance> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface **ElectricPermittivity**

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface ElectricPermittivity  
extends Quantity
```

This interface represents how an electric field affects, and is affected by a dielectric medium. The system unit for this quantity is "F/m" (farads per meter).

See Also:

[Wikipedia's Electric Permittivity](#)

---

Field Summary		Page
<a href="#">Unit&lt;ElectricPermittivity&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	78

### Field Detail

#### UNIT

```
public static final Unit<ElectricPermittivity> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface **ElectricPotential**

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface ElectricPotential
extends Quantity
```

This interface represents an electric potential or electromotive force. The system unit for this quantity is "V" (Volt).

---

Field Summary		Page
<a href="#">Unit&lt;ElectricPotential&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	79

### Field Detail

#### UNIT

```
public static final Unit<ElectricPotential> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface ElectricResistance

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface ElectricResistance  
extends Quantity
```

This interface represents an electric resistance. The system unit for this quantity is "Ohm" ( $\Omega$ ).

---

Field Summary		Page
<a href="#">Unit&lt;ElectricResistance&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	80

### Field Detail

#### UNIT

```
public static final Unit<ElectricResistance> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Energy

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Energy
extends Quantity
```

This interface represents the capacity of a physical system to do work. The system unit for this quantity "J" (Joule).

---

Field Summary		Page
<a href="#">Unit&lt;Energy&gt;</a> <b>UNIT</b>	Holds the SI unit (Système International d'Unités) for this quantity.	81

### Field Detail

#### UNIT

```
public static final Unit<Energy> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Force

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Force
extends Quantity
```

This interface represents a quantity that tends to produce an acceleration of a body in the direction of its application. The system unit for this quantity is "N" (Newton).

---

Field Summary		Page
<a href="#">Unit&lt;Force&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	82

### Field Detail

#### UNIT

```
public static final Unit<Force> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Frequency

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Frequency
extends Quantity
```

This interface represents the number of times a specified phenomenon occurs within a specified interval. The system unit for this quantity is "Hz" (Hertz).

---

Field Summary		Page
<a href="#">Unit&lt;Frequency&gt;</a>	<a href="#">UNIT</a>	83
Holds the SI unit (Système International d'Unités) for this quantity.		

### Field Detail

#### UNIT

```
public static final Unit<Frequency> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Illuminance

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Illuminance  
extends Quantity
```

This interface represents an illuminance. The system unit for this quantity is "lx" (lux).

---

Field Summary		Page
<a href="#">Unit&lt;Illuminance&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	84

### Field Detail

#### UNIT

```
public static final Unit<Illuminance> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface IonizingRadiation

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface IonizingRadiation  
extends Quantity
```

This interface represents the quantity of subatomic particles or electromagnetic waves that are energetic enough to detach electrons from atoms or molecules, ionizing them. The system unit for this quantity is "C/kg ("coulomb per kilogram).

See Also:

[Wikipedia's Ionizing Radiation](#)

---

Field Summary		Page
<a href="#">Unit&lt;IonizingRadiation&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	85

### Field Detail

#### UNIT

```
public static final Unit<IonizingRadiation> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface KinematicViscosity

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface KinematicViscosity  
extends Quantity
```

This interface represents the diffusion of momentum. The system unit for this quantity is "m2/s".

See Also:

[Wikipedia: Viscosity](#)

---

Field Summary		Page
<a href="#">Unit&lt;KinematicViscosity&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	86

### Field Detail

#### UNIT

```
public static final Unit<KinematicViscosity> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Length

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Length  
extends Quantity
```

This interface represents the extent of something along its greatest dimension or the extent of space between two objects or places. The system unit for this quantity is "m" (metre).

---

Field Summary		Page
<a href="#">Unit&lt;Length&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	87

### Field Detail

#### UNIT

```
public static final Unit<Length> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Luminance

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Luminance  
extends Quantity
```

This interface represents the luminous intensity per unit area of light traveling in a given direction. The system unit for this quantity is "cd/m2" (candela per square meter).

---

Field Summary		Page
<a href="#">Unit&lt;Luminance&gt;</a>	<a href="#">UNIT</a>	88
Holds the SI unit (Système International d'Unités) for this quantity.		

### Field Detail

#### UNIT

```
public static final Unit<Luminance> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface LuminousFlux

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface LuminousFlux
extends Quantity
```

This interface represents a luminous flux. The system unit for this quantity is "lm" (lumen).

---

Field Summary		Page
<a href="#">Unit&lt;LuminousFlux&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	89

### Field Detail

#### UNIT

```
public static final Unit<LuminousFlux> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface LuminousIntensity

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface LuminousIntensity
extends Quantity
```

This interface represents the luminous flux density per solid angle as measured in a given direction relative to the emitting source. The system unit for this quantity is "cd" (candela).

---

Field Summary		Page
<a href="#">Unit&lt;LuminousIntensity&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	90

### Field Detail

#### UNIT

```
public static final Unit<LuminousIntensity> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface MagneticFieldStrength

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface MagneticFieldStrength  
extends Quantity
```

This interface represents a magnetic field strength. The system unit for this quantity is "A/m" (ampere per meter).

---

Field Summary		Page
<a href="#">Unit&lt;MagneticFieldStrength&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	91

### Field Detail

#### UNIT

```
public static final Unit<MagneticFieldStrength> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface MagneticFlux

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface MagneticFlux
extends Quantity
```

This interface represents a magnetic flux. The system unit for this quantity is "Wb" (Weber).

---

Field Summary		Page
<a href="#">Unit&lt;MagneticFlux&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	92

### Field Detail

#### UNIT

```
public static final Unit<MagneticFlux> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface MagneticFluxDensity

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface MagneticFluxDensity
extends Quantity
```

This interface represents a magnetic flux density. The system unit for this quantity is "T" (Tesla).

---

Field Summary		Page
<a href="#">Unit&lt;MagneticFluxDensity&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	93

### Field Detail

#### UNIT

```
public static final Unit<MagneticFluxDensity> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface MagneticPermeability

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface MagneticPermeability  
extends Quantity
```

This interface represents the degree of magnetization of a material that responds linearly to an applied magnetic field. The system unit for this quantity is "H/m" (henry per meter).

See Also:

[Wikipedia's Permeability](#)

---

Field Summary		Page
<a href="#">Unit&lt;MagneticPermeability&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	94

### Field Detail

#### UNIT

```
public static final Unit<MagneticPermeability> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface MagnetomotiveForce

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

```
public interface MagnetomotiveForce
extends Quantity
```

This interface represents a force that produces magnetic flux. The system unit for this quantity is "At" (ampere-turn).

See Also:

[Wikipedia's Magnetomotive Force](#)

Field Summary		Page
<a href="#">Unit&lt;MagnetomotiveForce&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	95

### Field Detail

#### UNIT

```
public static final Unit<MagnetomotiveForce> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Mass

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Mass
extends Quantity
```

This interface represents the measure of the quantity of matter that a body or an object contains. The mass of the body is not dependent on gravity and therefore is different from but proportional to its weight. The system unit for this quantity is "kg" (kilogram).

---

Field Summary		Page
<a href="#">Unit&lt;Mass&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	96

### Field Detail

#### UNIT

```
public static final Unit<Mass> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface MassFlowRate

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface MassFlowRate  
extends Quantity
```

This interface represents the movement of mass per time. The system unit for this quantity is "kg/s" (kilogram per second).

See Also:

[Wikipedia: Mass Flow Rate](#)

---

Field Summary		Page
<a href="#">Unit&lt;MassFlowRate&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	97

### Field Detail

#### UNIT

```
public static final Unit<MassFlowRate> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Power

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Power
extends Quantity
```

This interface represents the rate at which work is done. The system unit for this quantity is "W" (Watt).

---

Field Summary		Page
<a href="#">Unit&lt;Power&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	98

## Field Detail

### UNIT

```
public static final Unit<Power> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Pressure

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Pressure
extends Quantity
```

This interface represents a force applied uniformly over a surface. The system unit for this quantity is "Pa" (Pascal).

---

Field Summary		Page
<a href="#">Unit&lt;Pressure&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	99

### Field Detail

#### UNIT

```
public static final Unit<Pressure> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Quantity

[javax.measure.quantity](#)

### All Known Subinterfaces:

[Acceleration](#), [Action](#), [AmountOfSubstance](#), [Angle](#), [AngularAcceleration](#), [AngularVelocity](#), [Area](#), [CatalyticActivity](#), [DataAmount](#), [DataRate](#), [Dimensionless](#), [Duration](#), [DynamicViscosity](#), [ElectricCapacitance](#), [ElectricCharge](#), [ElectricConductance](#), [ElectricCurrent](#), [ElectricInductance](#), [ElectricPermittivity](#), [ElectricPotential](#), [ElectricResistance](#), [Energy](#), [Force](#), [Frequency](#), [Illuminance](#), [IonizingRadiation](#), [KinematicViscosity](#), [Length](#), [Luminance](#), [LuminousFlux](#), [LuminousIntensity](#), [MagneticFieldStrength](#), [MagneticFlux](#), [MagneticFluxDensity](#), [MagneticPermeability](#), [MagnetomotiveForce](#), [Mass](#), [MassFlowRate](#), [Power](#), [Pressure](#), [RadiationDoseAbsorbed](#), [RadiationDoseEffective](#), [RadioactiveActivity](#), [SolidAngle](#), [Temperature](#), [Torque](#), [Velocity](#), [Volume](#), [VolumetricDensity](#), [VolumetricFlowRate](#), [Wavenumber](#)

---

```
public interface Quantity
```

This interface represents any type of quantitative properties or attributes of thing. Mass, time, distance, heat, and angular separation are among the familiar examples of quantitative properties.

Distinct quantities have usually different physical dimensions; although it is not required nor necessary, for example [Torque](#) and [Energy](#) have same dimension but are of different nature (vector for torque, scalar for energy).

### See Also:

[Wikipedia: Quantity](#), [Wikipedia: Dimensional Analysis](#)

## Interface RadiationDoseAbsorbed

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface RadiationDoseAbsorbed
extends Quantity
```

This interface represents the amount of energy deposited per unit of mass. The system unit for this quantity is "Gy" (Gray).

---

Field Summary		Page
<a href="#">Unit&lt;RadiationDoseAbsorbed&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	101

### Field Detail

#### UNIT

```
public static final Unit<RadiationDoseAbsorbed> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface RadiationDoseEffective

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface RadiationDoseEffective
extends Quantity
```

This interface represents the effective (or "equivalent") dose of radiation received by a human or some other living organism. The system unit for this quantity is "Sv" (Sievert).

---

Field Summary		Page
<a href="#">Unit&lt;RadiationDoseEffective&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	102

### Field Detail

#### UNIT

```
public static final Unit<RadiationDoseEffective> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface RadioactiveActivity

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface RadioactiveActivity  
extends Quantity
```

This interface represents a radioactive activity. The system unit for this quantity is "Bq" (Becquerel).

---

Field Summary		Page
<a href="#">Unit&lt;RadioactiveActivity&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	103

### Field Detail

#### UNIT

```
public static final Unit<RadioactiveActivity> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface SolidAngle

[javax.measure.quantity](#)

All Superinterfaces:

[Dimensionless](#), [Quantity](#)

---

```
public interface SolidAngle
extends Dimensionless
```

This interface represents the angle formed by three or more planes intersecting at a common point. The system unit for this quantity is "sr" (steradian). This quantity is dimensionless.

---

Field Summary		Page
<a href="#">Unit&lt;SolidAngle&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	104

### Field Detail

#### UNIT

```
public static final Unit<SolidAngle> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Temperature

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Temperature
extends Quantity
```

This class represents the degree of hotness or coldness of a body or an environment. The system unit for this quantity is "K" (Kelvin).

---

Field Summary		Page
<a href="#">Unit&lt;Temperature&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	105

### Field Detail

#### UNIT

```
public static final Unit<Temperature> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Torque

[javax.measure.quantity](#)

### All Superinterfaces:

[Quantity](#)

---

```
public interface Torque
extends Quantity
```

This interface represents the moment of a force. The system unit for this quantity is "N.m" (Newton-Metre).

Note: The Newton-metre ("N.m") is also a way of expressing a Joule (unit of energy). However, torque is not energy. So, to avoid confusion, we will use the units "N.m" for torque and not "J". This distinction occurs due to the scalar nature of energy and the vector nature of torque.

---

Field Summary		Page
<a href="#">Unit&lt;Torque&gt;</a> <b>UNIT</b>	Holds the SI unit (Système International d'Unités) for this quantity.	106

## Field Detail

### UNIT

```
public static final Unit<Torque> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Velocity

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

```
public interface Velocity
extends Quantity
```

This interface represents a distance traveled divided by the time of travel. The system unit for this quantity is "m/s" (metre per second).

Field Summary		Page
<a href="#">Unit&lt;Velocity&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	107

### Field Detail

#### UNIT

```
public static final Unit<Velocity> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Volume

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface Volume
extends Quantity
```

This interface represents the amount of space occupied by a three-dimensional object or region of space, expressed in cubic units. The system unit for this quantity is "m3" (cubic metre).

---

Field Summary		Page
<a href="#">Unit&lt;Volume&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	108

### Field Detail

#### UNIT

```
public static final Unit<Volume> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface VolumetricDensity

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface VolumetricDensity
extends Quantity
```

This interface represents a mass per unit volume of a substance under specified conditions of pressure and temperature. The system unit for this quantity is "kg/m3" (kilogram per cubic metre).

---

Field Summary		Page
<a href="#">Unit&lt;VolumetricDensity&gt;</a>	<a href="#">UNIT</a> Holds the SI unit (Système International d'Unités) for this quantity.	109

### Field Detail

#### UNIT

```
public static final Unit<VolumetricDensity> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface VolumetricFlowRate

[javax.measure.quantity](#)

All Superinterfaces:

[Quantity](#)

---

```
public interface VolumetricFlowRate  
extends Quantity
```

This interface represents the volume of fluid passing a point in a system per unit of time. The system unit for this quantity is "m3/s" (cubic metre per second).

See Also:

[Wikipedia: Volumetric Flow Rate](#)

---

Field Summary		Page
<a href="#">Unit&lt;VolumetricFlowRate&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	110

### Field Detail

#### UNIT

```
public static final Unit<VolumetricFlowRate> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Interface Wavenumber

[javax.measure.quantity](#)

**All Superinterfaces:**

[Quantity](#)

---

```
public interface Wavenumber  
extends Quantity
```

This interface represents a wave property inversely related to wavelength. The system unit for this quantity is "1/m" (reciprocal meters).

**See Also:**

[Wikipedia's Wavenumber](#)

---

Field Summary		Page
<a href="#">Unit&lt;Wavenumber&gt;</a>	<b>UNIT</b> Holds the SI unit (Système International d'Unités) for this quantity.	111

### Field Detail

#### UNIT

```
public static final Unit<Wavenumber> UNIT
```

Holds the SI unit (Système International d'Unités) for this quantity.

## Package javax.measure.unit

Provides support for programatic unit handling.

See:

[Description](#)

Interface Summary		Page
<a href="#">Dimension.Model</a>	This interface represents the mapping between <a href="#">base units</a> and <a href="#">dimensions</a> .	133

Class Summary		Page
<a href="#">AlternateUnit</a>	This class represents the units used in expressions to distinguish between quantities of a different nature but of the same dimensions.	114
<a href="#">AnnotatedUnit</a>	This class represents an annotated unit.	117
<a href="#">BaseUnit</a>	This class represents the building blocks on top of which all others units are created.	120
<a href="#">CompoundUnit</a>	This class represents the multi-radix units (such as "hour:min:sec").	123
<a href="#">DerivedUnit</a>	This class identifies the units created by combining or transforming other units.	126
<a href="#">Dimension</a>	This class represents the dimension of an unit.	127
<a href="#">NonSI</a>	This class contains units that are not part of the International System of Units, that is, they are outside the SI, but are important and widely used.	135
<a href="#">NonSI.BinaryPrefix</a>	Inner class holding binary prefixes.	152
<a href="#">NonSI.IndianPrefix</a>	Inner class holding prefixes used today in India, Pakistan, Bangladesh, Nepal and Myanmar (Burma); based on grouping by two decimal places, rather than the three decimal places common in most parts of the world.	154
<a href="#">ProductUnit</a>	This class represents units formed by the product of rational powers of existing units.	158
<a href="#">SI</a>	This class contains SI (Système International d'Unités) base units, and derived units.	162
<a href="#">SI.MetricPrefix</a>	Inner class holding prefixes used by the SI system.	171
<a href="#">SystemOfUnits</a>	This class represents a system of units, it groups units together for historical or cultural reasons.	177
<a href="#">TransformedUnit</a>	This class represents the units derived from other units using <a href="#">converters</a> .	178
<a href="#">UCUM</a>	This class contains the units ( <a href="#">SI</a> and <a href="#">NonSI</a> ) as defined in the <a href="#">Uniform Code for Units of Measure</a> .	181
<a href="#">Unit</a>	This class represents a determinate <a href="#">quantity</a> (as of length, time, heat, or value) adopted as a standard of measurement.	221
<a href="#">UnitFormat</a>	This class provides the interface for formatting and parsing <a href="#">units</a> .	230

## Package javax.measure.unit Description

Provides support for programatic unit handling.

### Standard/NonStandard Units

Standard units and prefixes are provided by the [SI](#) class (Système International d'Unités) and about 50 non-standard units are available through the [NonSI](#) class.

## Usage examples:

```
import javax.measure.Scalar;
import javax.measure.Measure;
import javax.measure.unit.*;
import javax.measure.quantity.*;
import static javax.measure.unit.SI.*;
import static javax.measure.unit.NonSI.*;
import static javax.measure.unit.Dimension.*;
public class Main {
    public void main(String[] args) {

        // Conversion between units.
        System.out.println(KILO(METRE).getConverterTo(MILE).convert(10));

        // Retrieval of the system unit (identifies the measurement type).
        System.out.println(REVOLUTION.divide(MINUTE).getSystemUnit());

        // Dimension checking (allows/disallows conversions)
        System.out.println(ELECTRON_VOLT.isCompatible(WATT.times(HOUR)));

        // Retrieval of the unit dimension (depends upon the current model).
        System.out.println(ELECTRON_VOLT.getDimension());
    }
}

> 6.2137119223733395
> rad/s
> true
> [L]2. [M] / [T]2
```

## Unit Parameterization

Units are parameterized (<Q extends [Quantity](#)>) to enforce compile-time checks of units/measures consistency, for example:

```
Unit<Duration> MINUTE = SECOND.times(60); // Ok.
Unit<Duration> MINUTE = METRE.times(60); // Compile error.

Unit<Pressure> HECTOPASCAL = HECTO(PASCAL); // Ok.
Unit<Pressure> HECTOPASCAL = HECTO(NEWTON); // Compile error.

Measurable<Duration> duration = Measure.valueOf(2, MINUTE); // Ok.
Measurable<Duration> duration = Measure.valueOf(2, CELSIUS); // Compile error.

long milliseconds = duration.longValue(MILLI(SECOND)); // Ok.
long milliseconds = duration.longValue(POUND); // Compile error.
```

Runtime checks of dimension consistency can be done for more complex cases.

```
Unit<Area> SQUARE_FOOT = FOOT.times(FOOT).asType(Area.class); // Ok.
Unit<Area> SQUARE_FOOT = FOOT.times(KELVIN).asType(Area.class); // Runtime error.

Unit<Temperature> KELVIN = Unit.valueOf("K").asType(Temperature.class); // Ok.
Unit<Temperature> KELVIN = Unit.valueOf("kg").asType(Temperature.class); // Runtime error
```

## Class **AlternateUnit**

[javax.measure.unit](#)

```
java.lang.Object
├─ javax.measure.unit.Unit<Q>
│   └─ javax.measure.unit.DerivedUnit<Q>
│       └─ javax.measure.unit.AlternateUnit
```

### All Implemented Interfaces:

Serializable

```
final public class AlternateUnit
extends DerivedUnit<Q>
```

This class represents the units used in expressions to distinguish between quantities of a different nature but of the same dimensions.

Instances of this class are created through the [Unit.alternate\(String\)](#) method.

Method Summary		Page
boolean	<a href="#">equals</a> (Object that) Indicates if the specified unit can be considered equals to the one specified.	115
<a href="#">UnitConverter</a>	<a href="#">getConverterToSI</a> () Returns the converter to the standard unit.	115
<a href="#">Unit</a> <?>	<a href="#">getParent</a> () Returns the parent unit from which this alternate unit is derived (a system unit itself).	114
String	<a href="#">getSymbol</a> () Returns the symbol for this alternate unit.	114
int	<a href="#">hashCode</a> () Returns the hash code for this unit.	115
<a href="#">Unit</a> <Q>	<a href="#">toSI</a> () Returns the standard unit from which this unit is derived.	115

## Method Detail

### **getSymbol**

```
public final String getSymbol ()
```

Returns the symbol for this alternate unit.

#### Returns:

this alternate unit symbol.

### **getParent**

```
public final Unit<?> getParent ()
```

Returns the parent unit from which this alternate unit is derived (a system unit itself).

**Returns:**  
the parent of the alternate unit.

---

## toSI

```
public final Unit<Q> toSI()
```

Returns the standard unit from which this unit is derived. The SI unit identifies the "type" of [quantity](#) for which this unit is employed. For example:

```
boolean isAngularVelocity(Unit<?> u) {  
    return u.toSI().equals(RADIAN.divide(SECOND));  
}  
assert (REVOLUTION.divide(MINUTE).isAngularVelocity());
```

**Overrides:**  
[toSI](#) in class [Unit](#)

**Returns:**  
the system unit this unit is derived from.

---

## getConverterToSI

```
public final UnitConverter getConverterToSI()
```

Returns the converter to the standard unit.

*Note: Having the same SI unit is not sufficient to ensure that a converter exists between the two units (e.g. °C/m and K/m).*

**Overrides:**  
[getConverterToSI](#) in class [Unit](#)

**Returns:**  
the unit converter from this unit to its standard unit.

---

## equals

```
public boolean equals(Object that)
```

Indicates if the specified unit can be considered equals to the one specified.

**Overrides:**  
[equals](#) in class [Unit](#)

**Parameters:**  
`that` - the object to compare to.

**Returns:**  
`true` if this unit is considered equal to that unit; `false` otherwise.

---

## hashCode

```
public int hashCode()
```

Returns the hash code for this unit.

**Overrides:**

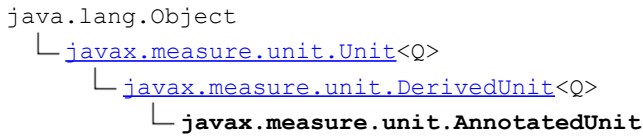
[hashCode](#) in class [Unit](#)

**Returns:**

this unit hashCode value.

# Class AnnotatedUnit

[javax.measure.unit](#)



**All Implemented Interfaces:**

Serializable

```

public class AnnotatedUnit
extends DerivedUnit<Q>
    
```

This class represents an annotated unit. It allows for unit specialization and annotation without changing the unit semantic. For example:

```

public class Size extends Measurable<Length> {
    private double meters;
    ...
    public static class Unit extends AnnotatedUnit<Length> {
        private Unit(javax.measure.unit.Unit<Length> realUnit, String annotation) {
            super(realUnit, annotation);
        }
        public static Size.Unit METER = new Size.Unit(SI.METRE, "size");
        // Equivalent to SI.METRE
        public static Size.Unit INCH = new Size.Unit(NonSI.INCH, "size");
        // Equivalent to NonSI.INCH
    }
}
    
```

Annotation are often written between curly braces behind units but they do not change, for example "%{vol}", "kg{total}", or "{RBC}" (for "red blood cells") are equivalent to "%", "kg", and "1" respectively.

Constructor Summary	Page
<a href="#">AnnotatedUnit</a> ( <a href="#">Unit</a> <Q> realUnit, String annotation) Creates an annotated unit for the specified unit.	118

Method Summary		Page
boolean	<a href="#">equals</a> (Object that) Indicates if the specified unit can be considered equals to the one specified.	119
String	<a href="#">getAnnotation</a> () Returns the annotation of this unit.	118
<a href="#">UnitConverter</a>	<a href="#">getConverterToSI</a> () Returns the converter to the standard unit.	119
<a href="#">Unit</a> <Q>	<a href="#">getRealUnit</a> () Returns the equivalent non-annotated unit.	118
int	<a href="#">hashCode</a> () Returns the hash code for this unit.	119
<a href="#">Unit</a> <Q>	<a href="#">toSI</a> () Returns the standard unit from which this unit is derived.	118

## Constructor Detail

### AnnotatedUnit

```
public AnnotatedUnit(Unit<Q> realUnit,  
                    String annotation)
```

Creates an annotated unit for the specified unit.

**Parameters:**

`realUnit` - the real unit.  
`annotation` - the annotation.

## Method Detail

### getAnnotation

```
public String getAnnotation()
```

Returns the annotation of this unit.

**Returns:**

the annotation of this unit.

---

### getRealUnit

```
public final Unit<Q> getRealUnit()
```

Returns the equivalent non-annotated unit.

**Returns:**

the real unit.

---

### toSI

```
public Unit<Q> toSI()
```

Returns the standard unit from which this unit is derived. The SI unit identifies the "type" of [quantity](#) for which this unit is employed. For example:

```
boolean isAngularVelocity(Unit<?> u) {  
    return u.toSI().equals(RADIAN.divide(SECOND));  
}  
assert (REVOLUTION.divide(MINUTE).isAngularVelocity());
```

**Overrides:**

[toSI](#) in class [Unit](#)

**Returns:**

the system unit this unit is derived from.

---

## getConverterToSI

```
public UnitConverter getConverterToSI()
```

Returns the converter to the standard unit.

*Note: Having the same SI unit is not sufficient to ensure that a converter exists between the two units (e.g. °C/m and K/m).*

**Overrides:**

[getConverterToSI](#) in class [Unit](#)

**Returns:**

the unit converter from this unit to its standard unit.

---

## equals

```
public boolean equals(Object that)
```

Indicates if the specified unit can be considered equals to the one specified.

**Overrides:**

[equals](#) in class [Unit](#)

**Parameters:**

`that` - the object to compare to.

**Returns:**

`true` if this unit is considered equal to that unit; `false` otherwise.

---

## hashCode

```
public int hashCode()
```

Returns the hash code for this unit.

**Overrides:**

[hashCode](#) in class [Unit](#)

**Returns:**

this unit hashcode value.

## Class **BaseUnit**

[javax.measure.unit](#)

```
java.lang.Object
├─ javax.measure.unit.Unit<Q>
│   └─ javax.measure.unit.BaseUnit
```

### All Implemented Interfaces:

Serializable

```
public class BaseUnit
extends Unit<Q>
```

This class represents the building blocks on top of which all others units are created. Base units are typically dimensionally independent. Although, the actual unit dimension is determinate by the current [model](#). Using the [standard](#) model, all seven SI base units are dimensionally independent.

This class defines the "standard base units" which includes SI base units and possibly others user-defined base units. It does not represent the base units of a specific [SystemOfUnits](#).

### See Also:

[Wikipedia: SI base unit](#)

Constructor Summary	Page
<a href="#">BaseUnit</a> (String symbol) Creates a base unit having the specified symbol.	120

Method Summary	Page
boolean <a href="#">equals</a> (Object that) Indicates if the specified unit can be considered equals to the one specified.	121
<a href="#">UnitConverter</a> <a href="#">getConverterToSI</a> () Returns the converter to the standard unit.	122
String <a href="#">getSymbol</a> () Returns the unique symbol for this base unit.	121
int <a href="#">hashCode</a> () Returns the hash code for this unit.	121
<a href="#">Unit</a> <Q> <a href="#">toSI</a> () Returns the standard unit from which this unit is derived.	121

## Constructor Detail

### BaseUnit

```
public BaseUnit(String symbol)
```

Creates a base unit having the specified symbol.

#### Parameters:

*symbol* - the symbol of this base unit.

#### Throws:

[IllegalArgumentException](#) - if the specified symbol is associated to a different unit.

## Method Detail

### getSymbol

```
public String getSymbol()
```

Returns the unique symbol for this base unit.

**Returns:**  
this base unit symbol.

---

### equals

```
public boolean equals(Object that)
```

Indicates if the specified unit can be considered equals to the one specified.

**Overrides:**  
[equals](#) in class [Unit](#)

**Parameters:**  
`that` - the object to compare to.

**Returns:**  
`true` if this unit is considered equal to that unit; `false` otherwise.

---

### hashCode

```
public int hashCode()
```

Returns the hash code for this unit.

**Overrides:**  
[hashCode](#) in class [Unit](#)

**Returns:**  
this unit hashcode value.

---

### toSI

```
public Unit<Q> toSI()
```

Returns the standard unit from which this unit is derived. The SI unit identifies the "type" of [quantity](#) for which this unit is employed. For example:

```
boolean isAngularVelocity(Unit<?> u) {  
    return u.toSI().equals(RADIAN.divide(SECOND));  
}  
assert(REVOLUTION.divide(MINUTE).isAngularVelocity());
```

**Overrides:**  
[toSI](#) in class [Unit](#)

**Returns:**  
the system unit this unit is derived from.

---

## getConverterToSI

```
public UnitConverter getConverterToSI ()
```

Returns the converter to the standard unit.

*Note: Having the same SI unit is not sufficient to ensure that a converter exists between the two units (e.g. °C/m and K/m).*

**Overrides:**

[getConverterToSI](#) in class [Unit](#)

**Returns:**

the unit converter from this unit to its standard unit.

# Class CompoundUnit

[javax.measure.unit](#)

```
java.lang.Object
├── javax.measure.unit.Unit<Q>
│   └── javax.measure.unit.DerivedUnit<Q>
│       └── javax.measure.unit.CompoundUnit
```

## All Implemented Interfaces:

Serializable

```
final public class CompoundUnit
extends DerivedUnit<Q>
```

This class represents the multi-radix units (such as "hour:min:sec"). Instances of this class are created using the [Unit.compound](#) method. Instances of this class are used mostly for [formatting](#) purpose.

Examples of compound units:

```
Unit<Duration> HOUR_MINUTE_SECOND = HOUR.compound(MINUTE).compound(SECOND);
Unit<Angle> DEGREE_MINUTE_ANGLE = DEGREE_ANGLE.compound(MINUTE_ANGLE);
Unit<Length> FOOT_INCH = FOOT.compound(INCH);
```

Method Summary		Page
boolean	<a href="#">equals</a> (Object that) Indicates if the specified unit can be considered equals to the one specified.	124
<a href="#">UnitConverter</a>	<a href="#">getConverterToSI</a> () Returns the converter to the standard unit.	125
<a href="#">Unit&lt;Q&gt;</a>	<a href="#">getHigh</a> () Returns the high unit(s) of this compound unit (can be a <a href="#">CompoundUnit</a> itself).	124
<a href="#">Unit&lt;Q&gt;</a>	<a href="#">getLow</a> () Returns the lowest unit or main unit of this compound unit (never a <a href="#">CompoundUnit</a> ).	123
int	<a href="#">hashCode</a> () Returns the hash code for this unit.	124
<a href="#">Unit&lt;Q&gt;</a>	<a href="#">toSI</a> () Returns the standard unit from which this unit is derived.	124
String	<a href="#">toString</a> () Overrides the <a href="#">default implementation</a> as compound units are not recognized by the standard UCUM format.	124

## Method Detail

### getLow

```
public Unit<Q> getLow()
```

Returns the lowest unit or main unit of this compound unit (never a [CompoundUnit](#)).

#### Returns:

the lower unit.

## getHigh

```
public Unit<Q> getHigh()
```

Returns the high unit(s) of this compound unit (can be a [CompoundUnit](#) itself).

**Returns:**  
the high unit(s).

---

## equals

```
public boolean equals(Object that)
```

Indicates if the specified unit can be considered equals to the one specified.

**Overrides:**  
[equals](#) in class [Unit](#)

**Parameters:**  
`that` - the object to compare to.

**Returns:**  
`true` if this unit is considered equal to that unit; `false` otherwise.

---

## hashCode

```
public int hashCode()
```

Returns the hash code for this unit.

**Overrides:**  
[hashCode](#) in class [Unit](#)

**Returns:**  
this unit hashcode value.

---

## toString

```
public String toString()
```

Overrides the [default implementation](#) as compound units are not recognized by the standard UCUM format.

**Overrides:**  
[toString](#) in class [Unit](#)

**Returns:**  
the textual representation of this compound unit.

---

## toSI

```
public Unit<Q> toSI()
```

Returns the standard unit from which this unit is derived. The SI unit identifies the "type" of [quantity](#) for which this unit is employed. For example:

```
boolean isAngularVelocity(Unit<?> u) {  
    return u.toSI().equals(RADIAN.divide(SECOND));  
}  
assert (REVOLUTION.divide(MINUTE).isAngularVelocity());
```

**Overrides:**

[toSI](#) in class [Unit](#)

**Returns:**

the system unit this unit is derived from.

---

## getConverterToSI

```
public final UnitConverter getConverterToSI()
```

Returns the converter to the standard unit.

*Note: Having the same SI unit is not sufficient to ensure that a converter exists between the two units (e.g. °C/m and K/m).*

**Overrides:**

[getConverterToSI](#) in class [Unit](#)

**Returns:**

the unit converter from this unit to its standard unit.

## Class DerivedUnit

[javax.measure.unit](#)

```
java.lang.Object
├─ javax.measure.unit.Unit<Q>
│   └─ javax.measure.unit.DerivedUnit
```

### All Implemented Interfaces:

Serializable

### Direct Known Subclasses:

[AlternateUnit](#), [AnnotatedUnit](#), [CompoundUnit](#), [ProductUnit](#), [TransformedUnit](#)

---

```
abstract public class DerivedUnit
extends Unit<Q>
```

This class identifies the units created by combining or transforming other units.

---

Constructor Summary		Page
protected	<a href="#">DerivedUnit</a> () Default constructor.	126

## Constructor Detail

### DerivedUnit

```
protected DerivedUnit ()
```

Default constructor.

# Class Dimension

[javax.measure.unit](#)

```
java.lang.Object
└─ javax.measure.unit.Dimension
```

**All Implemented Interfaces:**

Serializable

```
final public class Dimension
extends Object
implements Serializable
```

This class represents the dimension of an unit. Two units `u1` and `u2` are [compatible](#) if and only if `(u1.getDimension().equals(u2.getDimension()))`

Instances of this class are immutable.

**See Also:**

[Wikipedia: Dimensional Analysis](#)

Nested Class Summary		Page
static interface	<a href="#">Dimension.Model</a> This interface represents the mapping between <a href="#">base units</a> and <a href="#">dimensions</a> .	133

Field Summary		Page
static <a href="#">Dimension</a>	<a href="#">AMOUNT_OF_SUBSTANCE</a> Holds amount of substance dimension (N).	129
static <a href="#">Dimension</a>	<a href="#">ELECTRIC_CURRENT</a> Holds electric current dimension (I).	129
static <a href="#">Dimension</a>	<a href="#">LENGTH</a> Holds length dimension (L).	128
static <a href="#">Dimension</a>	<a href="#">LUMINOUS_INTENSITY</a> Holds luminous intensity dimension (J).	129
static <a href="#">Dimension</a>	<a href="#">MASS</a> Holds mass dimension (M).	128
static <a href="#">Dimension</a>	<a href="#">NONE</a> Holds dimensionless.	128
static <a href="#">Dimension</a>	<a href="#">TEMPERATURE</a> Holds temperature dimension (Q).	129
static <a href="#">Dimension</a>	<a href="#">TIME</a> Holds time dimension (T).	128

Constructor Summary		Page
<a href="#">Dimension</a> (char symbol)	Creates a new dimension associated to the specified symbol.	129

Method Summary		Page
<a href="#">Dimension</a>	<a href="#">divide</a> ( <a href="#">Dimension</a> that) Returns the quotient of this dimension with the one specified.	130

boolean	<a href="#">equals</a> (Object that) Indicates if the specified dimension is equals to the one specified.	131
<a href="#">Dimension</a>	<a href="#">getDimension</a> (int index) Returns the dimension element at the specified position.	130
int	<a href="#">getDimensionCount</a> () Returns the number of dimension elements in this dimension.	130
int	<a href="#">getDimensionPow</a> (int index) Returns the power exponent of the dimension element at the specified position.	131
int	<a href="#">getDimensionRoot</a> (int index) Returns the root exponent of the dimension element at the specified position.	131
static <a href="#">Dimension.Model</a>	<a href="#">getModel</a> () Returns the model used to determinate the units dimensions (default <a href="#">STANDARD</a> ).	132
int	<a href="#">hashCode</a> () Returns the hash code for this dimension.	132
<a href="#">Dimension</a>	<a href="#">pow</a> (int n) Returns this dimension raised to an exponent.	130
<a href="#">Dimension</a>	<a href="#">root</a> (int n) Returns the given root of this dimension.	130
static void	<a href="#">setModel</a> ( <a href="#">Dimension.Model</a> model) Sets the model used to determinate the units dimensions.	132
<a href="#">Dimension</a>	<a href="#">times</a> ( <a href="#">Dimension</a> that) Returns the product of this dimension with the one specified.	129
String	<a href="#">toString</a> () Returns the representation of this dimension.	131

## Field Detail

### NONE

```
public static final Dimension NONE
```

Holds dimensionless.

### LENGTH

```
public static final Dimension LENGTH
```

Holds length dimension (L).

### MASS

```
public static final Dimension MASS
```

Holds mass dimension (M).

### TIME

```
public static final Dimension TIME
```

Holds time dimension (T).

---

## ELECTRIC\_CURRENT

```
public static final Dimension ELECTRIC_CURRENT
```

Holds electric current dimension (I).

---

## TEMPERATURE

```
public static final Dimension TEMPERATURE
```

Holds temperature dimension (Q).

---

## AMOUNT\_OF\_SUBSTANCE

```
public static final Dimension AMOUNT_OF_SUBSTANCE
```

Holds amount of substance dimension (N).

---

## LUMINOUS\_INTENSITY

```
public static final Dimension LUMINOUS_INTENSITY
```

Holds luminous intensity dimension (J).

---

### Constructor Detail

#### Dimension

```
public Dimension(char symbol)
```

Creates a new dimension associated to the specified symbol.

**Parameters:**

`symbol` - the associated symbol.

### Method Detail

#### times

```
public final Dimension times(Dimension that)
```

Returns the product of this dimension with the one specified.

**Parameters:**

`that` - the dimension multiplicand.

**Returns:**

`this * that`

## divide

```
public final Dimension divide(Dimension that)
```

Returns the quotient of this dimension with the one specified.

**Parameters:**

that - the dimension divisor.

**Returns:**

this / that

---

## pow

```
public final Dimension pow(int n)
```

Returns this dimension raised to an exponent.

**Parameters:**

n - the exponent.

**Returns:**

the result of raising this dimension to the exponent.

---

## root

```
public final Dimension root(int n)
```

Returns the given root of this dimension.

**Parameters:**

n - the root's order.

**Returns:**

the result of taking the given root of this dimension.

**Throws:**

`ArithmeticException` - if `n == 0`.

---

## getDimensionCount

```
public int getDimensionCount()
```

Returns the number of dimension elements in this dimension.

**Returns:**

the number of dimension elements.

---

## getDimension

```
public Dimension getDimension(int index)
```

Returns the dimension element at the specified position.

**Parameters:**

`index` - the index of the dimension element to return.

**Returns:**

the dimension element at the specified position.

**Throws:**

`IndexOutOfBoundsException` - if `index` is out of range (`index < 0 || index >= getDimensionCount()`).

---

## getDimensionPow

```
public int getDimensionPow(int index)
```

Returns the power exponent of the dimension element at the specified position.

**Parameters:**

`index` - the index of the dimension element.

**Returns:**

the dimension power exponent at the specified position.

**Throws:**

`IndexOutOfBoundsException` - if `index` is out of range (`index < 0 || index >= getDimensionCount()`).

---

## getDimensionRoot

```
public int getDimensionRoot(int index)
```

Returns the root exponent of the dimension element at the specified position.

**Parameters:**

`index` - the index of the dimension element.

**Returns:**

the dimension root exponent at the specified position.

**Throws:**

`IndexOutOfBoundsException` - if `index` is out of range (`index < 0 || index >= getDimensionCount()`).

---

## toString

```
public String toString()
```

Returns the representation of this dimension.

**Overrides:**

`toString` in class `Object`

**Returns:**

the representation of this dimension.

---

## equals

```
public boolean equals(Object that)
```

Indicates if the specified dimension is equals to the one specified.

---

**Overrides:**

`equals` in class `Object`

**Parameters:**

`that` - the object to compare to.

**Returns:**

`true` if this dimension is equals to that dimension; `false` otherwise.

---

## hashCode

```
public int hashCode()
```

Returns the hash code for this dimension.

**Overrides:**

`hashCode` in class `Object`

**Returns:**

this dimension hashCode value.

---

## setModel

```
public static void setModel(Dimension.Model model)
```

Sets the model used to determinate the units dimensions.

**Parameters:**

`model` - the new model to be used when calculating unit dimensions.

---

## getModel

```
public static Dimension.Model getModel()
```

Returns the model used to determinate the units dimensions (default [STANDARD](#)).

**Returns:**

the model used when calculating unit dimensions.

# Interface Dimension.Model

[javax.measure.unit](#)

Enclosing class:  
[Dimension](#)

```
public static interface Dimension.Model
```

This interface represents the mapping between [base units](#) and [dimensions](#). Custom models may allow conversions not possible using the [standard](#) model. For example:

```
public static void main(String[] args) {
    Dimension.Model relativistic = new Dimension.Model() {
        RationalConverter metreToSecond
            = new RationalConverter(BigInteger.ONE, BigInteger.valueOf(299792458)); // 1/c

        public Dimension getDimension(BaseUnit unit) {
            if (unit.equals(METRE)) return Dimension.TIME;
            return Dimension.Model.STANDARD.getDimension(unit);
        }

        public UnitConverter getTransform(BaseUnit unit) {
            if (unit.equals(METRE)) return metreToSecond;
            return Dimension.Model.STANDARD.getTransform(unit);
        }
    };
    Dimension.setModel(relativistic);

    // Converts 1.0 GeV (energy) to kg (mass).
    System.out.println(Unit.valueOf("GeV").getConverterTo(KILOGRAM).convert(1.0));
}

> 1.7826617302520883E-27
```

Field Summary		Page
<a href="#">Dimension.Model</a>	<a href="#">STANDARD</a> Holds the standard model (default).	133

Method Summary		Page
<a href="#">Dimension</a>	<a href="#">getDimension</a> (BaseUnit<?> unit) Returns the dimension of the specified base unit (a dimension particular to the base unit if the base unit is not recognized).	134
<a href="#">UnitConverter</a>	<a href="#">getTransform</a> (BaseUnit<?> unit) Returns the normalization transform of the specified base unit ( <a href="#">IDENTITY</a> if the base unit is not recognized).	134

## Field Detail

### STANDARD

```
public static final Dimension.Model STANDARD
```

Holds the standard model (default).

## Method Detail

### getDimension

[Dimension](#) `getDimension(BaseUnit<?> unit)`

Returns the dimension of the specified base unit (a dimension particular to the base unit if the base unit is not recognized).

**Parameters:**

`unit` - the base unit for which the dimension is returned.

**Returns:**

the dimension of the specified unit.

---

### getTransform

[UnitConverter](#) `getTransform(BaseUnit<?> unit)`

Returns the normalization transform of the specified base unit ([IDENTITY](#) if the base unit is not recognized).

**Parameters:**

`unit` - the base unit for which the transform is returned.

**Returns:**

the normalization transform.

# Class NonSI

[javax.measure.unit](#)

```
java.lang.Object
├─ javax.measure.unit.SystemOfUnits
│   └─ javax.measure.unit.NonSI
```

```
final public class NonSI
extends SystemOfUnits
```

This class contains units that are not part of the International System of Units, that is, they are outside the SI, but are important and widely used.

Nested Class Summary		Page
static class	<a href="#">NonSI.BinaryPrefix</a> Inner class holding binary prefixes.	152
static class	<a href="#">NonSI.IndianPrefix</a> Inner class holding prefixes used today in India, Pakistan, Bangladesh, Nepal and Myanmar (Burma); based on grouping by two decimal places, rather than the three decimal places common in most parts of the world.	154

Field Summary		Page
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">ANGSTROM</a> A unit of length equal to 1E-10 m (standard name Å...).	140
static <a href="#">Unit&lt;Area&gt;</a>	<a href="#">ARE</a> A unit of area equal to 100 mÅ² (standard name a ).	146
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">ASTRONOMICAL_UNIT</a> A unit of length equal to the average distance from the center of the Earth to the center of the Sun (standard name ua).	141
static <a href="#">Unit&lt;Pressure&gt;</a>	<a href="#">ATMOSPHERE</a> A unit of pressure equal to the average pressure of the Earth's atmosphere at sea level (standard name atm).	148
static <a href="#">Unit&lt;AmountOfSubstance&gt;</a>	<a href="#">ATOM</a> A unit of amount of substance equals to one atom (standard name atom).	140
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">ATOMIC_MASS</a> A unit of mass equal to 1/12 the mass of the carbon-12 atom (standard name u).	143
static <a href="#">Unit&lt;Pressure&gt;</a>	<a href="#">BAR</a> A unit of pressure equal to 100 kPa (standard name bar).	148
static <a href="#">Unit&lt;DataAmount&gt;</a>	<a href="#">BYTE</a> A unit of data amount equal to 8 <a href="#">SI.BIT</a> (Binary TErms, standard name byte).	146
static <a href="#">Unit&lt;Velocity&gt;</a>	<a href="#">C</a> A unit of velocity relative to the speed of light (standard name c).	146
static <a href="#">Unit&lt;Angle&gt;</a>	<a href="#">CENTIRADIAN</a> A unit of angle equal to 0.01 <a href="#">SI.RADIAN</a> (standard name centiradian).	145
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">COMPUTER_POINT</a> Equivalent <a href="#">PIXEL</a>	141

static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">CUBIC_INCH</a> A unit of volume equal to one cubic inch (in <sup>3</sup> ).	149
static <a href="#">Unit&lt;RadioactiveActivity&gt;</a>	<a href="#">CURIE</a> A unit of radioactive activity equal to the activity of a gram of radium (standard name Ci).	149
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">DAY</a> A unit of duration equal to 24 <a href="#">HOUR</a> (standard name d).	142
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">DAY_SIDEREAL</a> A unit of duration equal to the time required for a complete rotation of the earth in reference to any star or to the vernal equinox at the meridian, equal to 23 hours, 56 minutes, 4.09 seconds (standard name day_sidereal).	142
static <a href="#">Unit&lt;Dimensionless&gt;</a>	<a href="#">DECIBEL</a> A logarithmic unit used to describe a ratio (standard name dB).	139
static <a href="#">Unit&lt;Angle&gt;</a>	<a href="#">DEGREE_ANGLE</a> A unit of angle equal to 1/360 <a href="#">REVOLUTION</a> (standard name deg).	144
static <a href="#">Unit&lt;Force&gt;</a>	<a href="#">DYNE</a> A unit of force equal to 1E-5 N (standard name dyn).	147
static <a href="#">Unit&lt;ElectricCharge&gt;</a>	<a href="#">E</a> A unit of electric charge equal to the charge on one electron (standard name e).	144
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">ELECTRON_MASS</a> A unit of mass equal to the mass of the electron (standard name m <sub>e</sub> ).	143
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">ELECTRON_VOLT</a> A unit of energy equal to one electron-volt (standard name eV, also recognized keV, MeV, GeV).	147
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">ERG</a> A unit of energy equal to 1E-7 J (standard name erg).	147
static <a href="#">Unit&lt;Temperature&gt;</a>	<a href="#">FAHRENHEIT</a> A unit of temperature equal to degree Rankine minus 459.67 Å°R (standard name Å°F).	144
static <a href="#">Unit&lt;ElectricCharge&gt;</a>	<a href="#">FARADAY</a> A unit of electric charge equal to equal to the product of Avogadro's number (see <a href="#">SI.MOLE</a> ) and the charge (1 e) on a single electron (standard name F <sub>d</sub> ).	144
static <a href="#">Unit&lt;Velocity&gt;</a>	<a href="#">FEET_PER_SECOND</a> A unit of velocity expressing the number of <a href="#">feet</a> per <a href="#">second</a> .	145
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">FOOT</a> A unit of length equal to 0.3048 m (standard name ft).	140
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">FOOT_SURVEY_US</a> A unit of length equal to 1200/3937 m (standard name foot_survey_us).	140
static <a href="#">Unit&lt;ElectricCharge&gt;</a>	<a href="#">FRANKLIN</a> A unit of electric charge which exerts a force of one dyne on an equal charge at a distance of one centimeter (standard name F <sub>r</sub> ).	144
static <a href="#">Unit&lt;Acceleration&gt;</a>	<a href="#">G</a> A unit of acceleration equal to the gravity at the earth's surface (standard name grav).	146
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">GALLON_DRY_US</a> A unit of volume equal to one US dry gallon.	150
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">GALLON_LIQUID_US</a> A unit of volume equal to one US gallon, Liquid Unit.	149

static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">GALLON_UK</a> A unit of volume equal to 4.546 09 <a href="#">LITRE</a> (standard name gal_uk).	150
static <a href="#">Unit&lt;MagneticFluxDensity&gt;</a>	<a href="#">GAUSS</a> A unit of magnetic flux density equal 1000 A/m (standard name G).	147
static <a href="#">Unit&lt;ElectricCurrent&gt;</a>	<a href="#">GILBERT</a> A unit of electric charge equal to the centimeter-gram-second electromagnetic unit of magnetomotive force, equal to 10/4 ampere-turn (standard name Gi).	146
static <a href="#">Unit&lt;Angle&gt;</a>	<a href="#">GRADE</a> A unit of angle measure equal to 1/400 <a href="#">REVOLUTION</a> (standard name grade ).	145
static <a href="#">Unit&lt;Area&gt;</a>	<a href="#">HECTARE</a> A unit of area equal to 100 <a href="#">ARE</a> (standard name ha).	146
static <a href="#">Unit&lt;Power&gt;</a>	<a href="#">HORSEPOWER</a> A unit of power equal to the power required to raise a mass of 75 kilograms at a velocity of 1 meter per second (metric, standard name hp).	148
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">HOUR</a> A unit of duration equal to 60 <a href="#">MINUTE</a> (standard name h).	142
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">INCH</a> A unit of length equal to 0.0254 m (standard name in).	140
static <a href="#">Unit&lt;Pressure&gt;</a>	<a href="#">INCH_OF_MERCURY</a> A unit of pressure equal to the pressure exerted at the Earth's surface by a column of mercury 1 inch high (standard name inHg).	148
static <a href="#">Unit&lt;Force&gt;</a>	<a href="#">KILOGRAM_FORCE</a> A unit of force equal to 9.80665 N (standard name kgf).	147
static <a href="#">Unit&lt;Velocity&gt;</a>	<a href="#">KILOMETRES_PER_HOUR</a> A unit of velocity expressing the number of <a href="#">SI.KILOMETRE</a> per <a href="#">hour</a> .	145
static <a href="#">Unit&lt;Velocity&gt;</a>	<a href="#">KNOT</a> A unit of velocity expressing the number of <a href="#">nautical miles per hour</a> (abbreviation kn).	146
static <a href="#">Unit&lt;Illuminance&gt;</a>	<a href="#">LAMBERT</a> A unit of illuminance equal to 1E4 Lx (standard name La).	147
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">LIGHT_YEAR</a> A unit of length equal to the distance that light travels in one year through a vacuum (standard name ly).	141
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">LITRE</a> A unit of volume equal to one cubic decimeter (default label L, also recognized µL, mL, cL, dL).	149
static <a href="#">Unit&lt;MagneticFlux&gt;</a>	<a href="#">MAXWELL</a> A unit of magnetic flux equal 1E-8 Wb (standard name Mx).	147
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">METRIC_TON</a> A unit of mass equal to 1000 kg (metric ton, standard name t).	143
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">MILE</a> A unit of length equal to 1609.344 m (standard name mi).	140
static <a href="#">Unit&lt;Velocity&gt;</a>	<a href="#">MILES_PER_HOUR</a> A unit of velocity expressing the number of international <a href="#">miles per hour</a> (abbreviation mph).	145
static <a href="#">Unit&lt;Pressure&gt;</a>	<a href="#">MILLIMETRE_OF_MERCURY</a> A unit of pressure equal to the pressure exerted at the Earth's surface by a column of mercury 1 millimeter high (standard name mmHg ).	148
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">MINUTE</a> A unit of duration equal to 60 s (standard name min).	142

static <a href="#">Unit&lt;Angle&gt;</a>	<a href="#">MINUTE_ANGLE</a> A unit of angle equal to 1/60 <a href="#">DEGREE_ANGLE</a> (standard name ' ).	145
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">NAUTICAL_MILE</a> A unit of length equal to 1852.0 m (standard name nmi).	140
static <a href="#">Unit&lt;DataAmount&gt;</a>	<a href="#">OCTET</a> Equivalent <a href="#">BYTE</a>	146
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">OUNCE</a> A unit of mass equal to 1 / 16 <a href="#">POUND</a> (standard name oz).	143
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">OUNCE_LIQUID_UK</a> A unit of volume equal to 1 / 160 <a href="#">GALLON_UK</a> (standard name oz_fl_uk).	150
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">OUNCE_LIQUID_US</a> A unit of volume equal to 1 / 128 <a href="#">GALLON_LIQUID_US</a> (standard name oz_fl).	149
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">PARSEC</a> A unit of length equal to the distance at which a star would appear to shift its position by one arcsecond over the course the time (about 3 months) in which the Earth moves a distance of <a href="#">ASTRONOMICAL_UNIT</a> in the direction perpendicular to the direction to the star (standard name pc).	141
static <a href="#">Unit&lt;Dimensionless&gt;</a>	<a href="#">PERCENT</a> A dimensionless unit equals to 0.01 (standard name %).	139
static <a href="#">Unit&lt;Dimensionless&gt;</a>	<a href="#">PI</a> A dimensionless unit equals to pi (standard name ĩe).	139
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">PIXEL</a> A unit of length equal to 1/72 <a href="#">INCH</a> (standard name pixel).	141
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">POINT</a> A unit of length equal to 0.013837 <a href="#">INCH</a> exactly (standard name pt).	141
static <a href="#">Unit&lt;DynamicViscosity&gt;</a>	<a href="#">POISE</a> A unit of dynamic viscosity equal to 1 g/(cm <sup>Å</sup> ·s) (cgs unit).	150
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">POUND</a> A unit of mass equal to 453.59237 grams (avoirdupois pound, standard name lb).	143
static <a href="#">Unit&lt;Force&gt;</a>	<a href="#">POUND_FORCE</a> A unit of force equal to <a href="#">POUND</a> · <a href="#">G</a> (standard name lbf).	148
static <a href="#">Unit&lt;RadiationDoseAbsorbed&gt;</a>	<a href="#">RAD</a> A unit of radiation dose absorbed equal to a dose of 0.01 joule of energy per kilogram of mass (J/kg) (standard name rd).	148
static <a href="#">Unit&lt;Temperature&gt;</a>	<a href="#">RANKINE</a> A unit of temperature equal to 5/9 <sup>Å</sup> K (standard name <sup>Å</sup> R).	144
static <a href="#">Unit&lt;RadiationDoseEffective&gt;</a>	<a href="#">REM</a> A unit of radiation dose effective equal to 0.01 Sv (standard name rem).	149
static <a href="#">Unit&lt;Angle&gt;</a>	<a href="#">REVOLUTION</a> A unit of angle equal to a full circle or 2π <a href="#">SI_RADIAN</a> (standard name rev).	144
static <a href="#">Unit&lt;?&gt;</a>	<a href="#">ROENTGEN</a> A unit used to measure the ionizing ability of radiation (standard name Roentgen).	150
static <a href="#">Unit&lt;RadioactiveActivity&gt;</a>	<a href="#">RUTHERFORD</a> A unit of radioactive activity equal to 1 million radioactive disintegrations per second (standard name Rd).	149
static <a href="#">Unit&lt;Angle&gt;</a>	<a href="#">SECOND_ANGLE</a> A unit of angle equal to 1/60 <a href="#">MINUTE_ANGLE</a> (standard name ").	145

static <a href="#">Unit&lt;SolidAngle&gt;</a>	<a href="#">SPHERE</a> A unit of solid angle equal to $4 \pi$ steradians (standard name <code>sphere</code> ).	149
static <a href="#">Unit&lt;KinematicViscosity&gt;</a>	<a href="#">STOKE</a> A unit of kinematic viscosity equal to $1 \text{ cm}^2/\text{s}$ (cgs unit).	150
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">TON_UK</a> A unit of mass equal to 2240 <a href="#">POUND</a> (long ton, standard name <code>ton_uk</code> ).	143
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">TON_US</a> A unit of mass equal to 2000 <a href="#">POUND</a> (short ton, standard name <code>ton_us</code> ).	143
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">WEEK</a> A unit of duration equal to 7 <a href="#">DAY</a> (standard name <code>week</code> ).	142
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">YARD</a> A unit of length equal to 0.9144 m (standard name <code>yd</code> ).	140
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">YEAR_CALENDAR</a> A unit of duration equal to 365 <a href="#">DAY</a> (standard name <code>year</code> ).	142
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">YEAR_JULIEN</a> The Julian year, as used in astronomy and other sciences, is a time unit defined as exactly 365.25 days.	143
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">YEAR_SIDEREAL</a> A unit of duration equal to one complete revolution of the earth about the sun, relative to the fixed stars, or 365 days, 6 hours, 9 minutes, 9.54 seconds (standard name <code>year_sidereal</code> ).	142

Method Summary		Page
static <a href="#">NonSI</a>	<a href="#">getInstance</a> () Returns the unique instance of this class.	150
Set< <a href="#">Unit</a> <?>>	<a href="#">getUnits</a> () Returns a read only view over the units defined in this class.	151

## Field Detail

### PI

public static final [Unit<Dimensionless>](#) **PI**

A dimensionless unit equals to  $\pi$  (standard name `ie`).

### PERCENT

public static final [Unit<Dimensionless>](#) **PERCENT**

A dimensionless unit equals to 0.01 (standard name `%`).

### DECIBEL

public static final [Unit<Dimensionless>](#) **DECIBEL**

A logarithmic unit used to describe a ratio (standard name `dB`).

## ATOM

public static final [Unit<AmountOfSubstance>](#) **ATOM**

A unit of amount of substance equals to one atom (standard name `atom`).

---

## FOOT

public static final [Unit<Length>](#) **FOOT**

A unit of length equal to 0.3048 m (standard name `ft`).

---

## FOOT\_SURVEY\_US

public static final [Unit<Length>](#) **FOOT\_SURVEY\_US**

A unit of length equal to 1200/3937 m (standard name `foot_survey_us`). See also: [foot](#)

---

## YARD

public static final [Unit<Length>](#) **YARD**

A unit of length equal to 0.9144 m (standard name `yd`).

---

## INCH

public static final [Unit<Length>](#) **INCH**

A unit of length equal to 0.0254 m (standard name `in`).

---

## MILE

public static final [Unit<Length>](#) **MILE**

A unit of length equal to 1609.344 m (standard name `mi`).

---

## NAUTICAL\_MILE

public static final [Unit<Length>](#) **NAUTICAL\_MILE**

A unit of length equal to 1852.0 m (standard name `nmi`).

---

## ANGSTROM

public static final [Unit<Length>](#) **ANGSTROM**

A unit of length equal to 1E-10 m (standard name `Å...`).

## ASTRONOMICAL\_UNIT

```
public static final Unit<Length> ASTRONOMICAL_UNIT
```

A unit of length equal to the average distance from the center of the Earth to the center of the Sun (standard name `ua`).

---

## LIGHT\_YEAR

```
public static final Unit<Length> LIGHT_YEAR
```

A unit of length equal to the distance that light travels in one year through a vacuum (standard name `ly`).

---

## PARSEC

```
public static final Unit<Length> PARSEC
```

A unit of length equal to the distance at which a star would appear to shift its position by one arcsecond over the course the time (about 3 months) in which the Earth moves a distance of [ASTRONOMICAL\\_UNIT](#) in the direction perpendicular to the direction to the star (standard name `pc`).

---

## POINT

```
public static final Unit<Length> POINT
```

A unit of length equal to 0.013837 [INCH](#) exactly (standard name `pt`).

**See Also:**

[PIXEL](#)

---

## PIXEL

```
public static final Unit<Length> PIXEL
```

A unit of length equal to  $1/72$  [INCH](#) (standard name `pixel`). It is the American point rounded to an even  $1/72$  inch.

**See Also:**

[POINT](#)

---

## COMPUTER\_POINT

```
public static final Unit<Length> COMPUTER_POINT
```

Equivalent [PIXEL](#)

---

## MINUTE

```
public static final Unit<Duration> MINUTE
```

A unit of duration equal to 60 s (standard name min).

---

## HOUR

```
public static final Unit<Duration> HOUR
```

A unit of duration equal to 60 [MINUTE](#) (standard name h).

---

## DAY

```
public static final Unit<Duration> DAY
```

A unit of duration equal to 24 [HOUR](#) (standard name d).

---

## DAY\_SIDEREAL

```
public static final Unit<Duration> DAY_SIDEREAL
```

A unit of duration equal to the time required for a complete rotation of the earth in reference to any star or to the vernal equinox at the meridian, equal to 23 hours, 56 minutes, 4.09 seconds (standard name day\_sidereal).

---

## WEEK

```
public static final Unit<Duration> WEEK
```

A unit of duration equal to 7 [DAY](#) (standard name week).

---

## YEAR\_CALENDAR

```
public static final Unit<Duration> YEAR_CALENDAR
```

A unit of duration equal to 365 [DAY](#) (standard name year).

---

## YEAR\_SIDEREAL

```
public static final Unit<Duration> YEAR_SIDEREAL
```

A unit of duration equal to one complete revolution of the earth about the sun, relative to the fixed stars, or 365 days, 6 hours, 9 minutes, 9.54 seconds (standard name year\_sidereal).

---

## YEAR\_JULIEN

public static final [Unit<Duration>](#) YEAR\_JULIEN

The Julian year, as used in astronomy and other sciences, is a time unit defined as exactly 365.25 days. This is the normal meaning of the unit "year" (symbol "a" from the Latin annus, annata) used in various scientific contexts.

---

## ATOMIC\_MASS

public static final [Unit<Mass>](#) ATOMIC\_MASS

A unit of mass equal to 1/12 the mass of the carbon-12 atom (standard name u).

---

## ELECTRON\_MASS

public static final [Unit<Mass>](#) ELECTRON\_MASS

A unit of mass equal to the mass of the electron (standard name me).

---

## POUND

public static final [Unit<Mass>](#) POUND

A unit of mass equal to 453.59237 grams (avoirdupois pound, standard name lb).

---

## OUNCE

public static final [Unit<Mass>](#) OUNCE

A unit of mass equal to 1 / 16 [POUND](#) (standard name oz).

---

## TON\_US

public static final [Unit<Mass>](#) TON\_US

A unit of mass equal to 2000 [POUND](#) (short ton, standard name ton\_us).

---

## TON\_UK

public static final [Unit<Mass>](#) TON\_UK

A unit of mass equal to 2240 [POUND](#) (long ton, standard name ton\_uk).

---

## METRIC\_TON

public static final [Unit<Mass>](#) METRIC\_TON

A unit of mass equal to 1000 kg (metric ton, standard name t).

---

## E

public static final [Unit<ElectricCharge>](#) **E**

A unit of electric charge equal to the charge on one electron (standard name e).

---

## FARADAY

public static final [Unit<ElectricCharge>](#) **FARADAY**

A unit of electric charge equal to equal to the product of Avogadro's number (see [SI.MOLE](#)) and the charge (1 e) on a single electron (standard name  $F_d$ ).

---

## FRANKLIN

public static final [Unit<ElectricCharge>](#) **FRANKLIN**

A unit of electric charge which exerts a force of one dyne on an equal charge at a distance of one centimeter (standard name  $F_r$ ).

---

## RANKINE

public static final [Unit<Temperature>](#) **RANKINE**

A unit of temperature equal to  $5/9 \text{ }^\circ\text{K}$  (standard name  $^\circ\text{R}$ ).

---

## FAHRENHEIT

public static final [Unit<Temperature>](#) **FAHRENHEIT**

A unit of temperature equal to degree Rankine minus  $459.67 \text{ }^\circ\text{R}$  (standard name  $^\circ\text{F}$ ).

**See Also:**

[RANKINE](#)

---

## REVOLUTION

public static final [Unit<Angle>](#) **REVOLUTION**

A unit of angle equal to a full circle or  $2\pi$ [SI.RADIAN](#) (standard name rev).

---

## DEGREE\_ANGLE

public static final [Unit<Angle>](#) **DEGREE\_ANGLE**

A unit of angle equal to  $1/360$  [REVOLUTION](#) (standard name deg).

---

## MINUTE\_ANGLE

```
public static final Unit<Angle> MINUTE_ANGLE
```

A unit of angle equal to  $1/60$  [DEGREE\\_ANGLE](#) (standard name ').

---

## SECOND\_ANGLE

```
public static final Unit<Angle> SECOND_ANGLE
```

A unit of angle equal to  $1/60$  [MINUTE\\_ANGLE](#) (standard name ").

---

## CENTIRADIAN

```
public static final Unit<Angle> CENTIRADIAN
```

A unit of angle equal to 0.01 [SI.RADIAN](#) (standard name centiradian).

---

## GRADE

```
public static final Unit<Angle> GRADE
```

A unit of angle measure equal to  $1/400$  [REVOLUTION](#) (standard name grade ).

---

## FEET\_PER\_SECOND

```
public static final Unit<Velocity> FEET_PER_SECOND
```

A unit of velocity expressing the number of [feet](#) per [second](#).

---

## MILES\_PER\_HOUR

```
public static final Unit<Velocity> MILES_PER_HOUR
```

A unit of velocity expressing the number of international [miles](#) per [hour](#) (abbreviation mph).

---

## KILOMETRES\_PER\_HOUR

```
public static final Unit<Velocity> KILOMETRES_PER_HOUR
```

A unit of velocity expressing the number of [SI.KILOMETRE](#) per [hour](#).

---

## KNOT

public static final [Unit<Velocity>](#) **KNOT**

A unit of velocity expressing the number of [nautical miles](#) per [hour](#) (abbreviation `kn`).

---

## C

public static final [Unit<Velocity>](#) **C**

A unit of velocity relative to the speed of light (standard name `c`).

---

## G

public static final [Unit<Acceleration>](#) **G**

A unit of acceleration equal to the gravity at the earth's surface (standard name `grav`).

---

## ARE

public static final [Unit<Area>](#) **ARE**

A unit of area equal to  $100 \text{ m}^2$  (standard name `a`).

---

## HECTARE

public static final [Unit<Area>](#) **HECTARE**

A unit of area equal to 100 [ARE](#) (standard name `ha`).

---

## BYTE

public static final [Unit<DataAmount>](#) **BYTE**

A unit of data amount equal to 8 [SI\\_BIT](#) (Binary Term, standard name `byte`).

---

## OCTET

public static final [Unit<DataAmount>](#) **OCTET**

Equivalent [BYTE](#)

---

## GILBERT

public static final [Unit<ElectricCurrent>](#) **GILBERT**

A unit of electric charge equal to the centimeter-gram-second electromagnetic unit of magnetomotive force,

equal to  $10/4$  ampere-turn (standard name Gi).

---

## ERG

public static final [Unit<Energy>](#) **ERG**

A unit of energy equal to  $1\text{E-}7$  J (standard name erg).

---

## ELECTRON\_VOLT

public static final [Unit<Energy>](#) **ELECTRON\_VOLT**

A unit of energy equal to one electron-volt (standard name eV, also recognized keV, MeV, GeV).

---

## LAMBERT

public static final [Unit<Illuminance>](#) **LAMBERT**

A unit of illuminance equal to  $1\text{E}4$  lx (standard name La).

---

## MAXWELL

public static final [Unit<MagneticFlux>](#) **MAXWELL**

A unit of magnetic flux equal  $1\text{E-}8$  Wb (standard name Mx).

---

## GAUSS

public static final [Unit<MagneticFluxDensity>](#) **GAUSS**

A unit of magnetic flux density equal  $1000$  A/m (standard name G).

---

## DYNE

public static final [Unit<Force>](#) **DYNE**

A unit of force equal to  $1\text{E-}5$  N (standard name dyn).

---

## KILOGRAM\_FORCE

public static final [Unit<Force>](#) **KILOGRAM\_FORCE**

A unit of force equal to  $9.80665$  N (standard name kgf).

---

## POUND\_FORCE

public static final [Unit<Force>](#) POUND\_FORCE

A unit of force equal to  $\text{POUND} \cdot \text{g}$  (standard name lbf).

---

## HORSEPOWER

public static final [Unit<Power>](#) HORSEPOWER

A unit of power equal to the power required to raise a mass of 75 kilograms at a velocity of 1 meter per second (metric, standard name hp).

---

## ATMOSPHERE

public static final [Unit<Pressure>](#) ATMOSPHERE

A unit of pressure equal to the average pressure of the Earth's atmosphere at sea level (standard name atm).

---

## BAR

public static final [Unit<Pressure>](#) BAR

A unit of pressure equal to 100 kPa (standard name bar).

---

## MILLIMETRE\_OF\_MERCURY

public static final [Unit<Pressure>](#) MILLIMETRE\_OF\_MERCURY

A unit of pressure equal to the pressure exerted at the Earth's surface by a column of mercury 1 millimeter high (standard name mmHg ).

---

## INCH\_OF\_MERCURY

public static final [Unit<Pressure>](#) INCH\_OF\_MERCURY

A unit of pressure equal to the pressure exerted at the Earth's surface by a column of mercury 1 inch high (standard name inHg).

---

## RAD

public static final [Unit<RadiationDoseAbsorbed>](#) RAD

A unit of radiation dose absorbed equal to a dose of 0.01 joule of energy per kilogram of mass (J/kg) (standard name rd).

---

## REM

```
public static final Unit<RadiationDoseEffective> REM
```

A unit of radiation dose effective equal to 0.01 Sv (standard name rem).

---

## CURIE

```
public static final Unit<RadioactiveActivity> CURIE
```

A unit of radioactive activity equal to the activity of a gram of radium (standard name Ci).

---

## RUTHERFORD

```
public static final Unit<RadioactiveActivity> RUTHERFORD
```

A unit of radioactive activity equal to 1 million radioactive disintegrations per second (standard name Rd).

---

## SPHERE

```
public static final Unit<SolidAngle> SPHERE
```

A unit of solid angle equal to  $4\pi$  steradians (standard name sphere).

---

## LITRE

```
public static final Unit<Volume> LITRE
```

A unit of volume equal to one cubic decimeter (default label L, also recognized  $\mu\text{L}$ , mL, cL, dL).

---

## CUBIC\_INCH

```
public static final Unit<Volume> CUBIC_INCH
```

A unit of volume equal to one cubic inch ( $\text{in}^3$ ).

---

## GALLON\_LIQUID\_US

```
public static final Unit<Volume> GALLON_LIQUID_US
```

A unit of volume equal to one US gallon, Liquid Unit. The U.S. liquid gallon is based on the Queen Anne or Wine gallon occupying 231 cubic inches (standard name gal).

---

## OUNCE\_LIQUID\_US

```
public static final Unit<Volume> OUNCE_LIQUID_US
```

A unit of volume equal to  $1 / 128$  [GALLON\\_LIQUID\\_US](#) (standard name `oz_fl`).

---

## GALLON\_DRY\_US

```
public static final Unit<Volume> GALLON_DRY_US
```

A unit of volume equal to one US dry gallon. (standard name `gallon_dry_us`).

---

## GALLON\_UK

```
public static final Unit<Volume> GALLON_UK
```

A unit of volume equal to  $4.546\ 09$  [LITRE](#) (standard name `gal_uk`).

---

## OUNCE\_LIQUID\_UK

```
public static final Unit<Volume> OUNCE_LIQUID_UK
```

A unit of volume equal to  $1 / 160$  [GALLON\\_UK](#) (standard name `oz_fl_uk`).

---

## POISE

```
public static final Unit<DynamicViscosity> POISE
```

A unit of dynamic viscosity equal to  $1\ \text{g} / (\text{cm} \cdot \text{s})$  (cgs unit).

---

## STOKE

```
public static final Unit<KinematicViscosity> STOKE
```

A unit of kinematic viscosity equal to  $1\ \text{cm}^2 / \text{s}$  (cgs unit).

---

## ROENTGEN

```
public static final Unit<?> ROENTGEN
```

A unit used to measure the ionizing ability of radiation (standard name `Roentgen`).

## Method Detail

### getInstance

```
public static NonSI getInstance()
```

Returns the unique instance of this class.

**Returns:**  
the NonSI instance.

---

## getUnits

```
public Set<Unit<?>> getUnits ()
```

Returns a read only view over the units defined in this class.

**Overrides:**  
[getUnits](#) in class [SystemOfUnits](#)

**Returns:**  
the collection of NonSI units.

## Class NonSI.BinaryPrefix

[javax.measure.unit](#)

```
java.lang.Object
└─ javax.measure.unit.NonSI.BinaryPrefix
```

Enclosing class:

[NonSI](#)

```
public static class NonSI.BinaryPrefix
extends Object
```

Inner class holding binary prefixes.

Method Summary		Page
static <a href="#">Unit</a> <Q>	<a href="#">EXBI</a> ( <a href="#">Unit</a> <Q> unit) Returns the specified unit multiplied by the factor $2^{60}$ (binary prefix).	153
static <a href="#">Unit</a> <Q>	<a href="#">GIBI</a> ( <a href="#">Unit</a> <Q> unit) Returns the specified unit multiplied by the factor $2^{30}$ (binary prefix).	153
static <a href="#">Unit</a> <Q>	<a href="#">KIBI</a> ( <a href="#">Unit</a> <Q> unit) Returns the specified unit multiplied by the factor $2^{10}$ (binary prefix).	152
static <a href="#">Unit</a> <Q>	<a href="#">MEBI</a> ( <a href="#">Unit</a> <Q> unit) Returns the specified unit multiplied by the factor $2^{20}$ (binary prefix).	152
static <a href="#">Unit</a> <Q>	<a href="#">PEBI</a> ( <a href="#">Unit</a> <Q> unit) Returns the specified unit multiplied by the factor $2^{50}$ (binary prefix).	153
static <a href="#">Unit</a> <Q>	<a href="#">TEBI</a> ( <a href="#">Unit</a> <Q> unit) Returns the specified unit multiplied by the factor $2^{40}$ (binary prefix).	153

## Method Detail

### KIBI

```
public static Unit<Q> KIBI (Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $2^{10}$  (binary prefix).

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1024)`.

### MEBI

```
public static Unit<Q> MEBI (Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $2^{20}$  (binary prefix).

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1048576).`

---

**GIBI**

`public static final Unit<Q> GIBI(Unit<Q> unit)`

Returns the specified unit multiplied by the factor  $2^{30}$  (binary prefix).

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1073741824).`

---

**TEBI**

`public static Unit<Q> TEBI(Unit<Q> unit)`

Returns the specified unit multiplied by the factor  $2^{40}$  (binary prefix).

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1099511627776L).`

---

**PEBI**

`public static Unit<Q> PEBI(Unit<Q> unit)`

Returns the specified unit multiplied by the factor  $2^{50}$  (binary prefix).

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1125899906842624L).`

---

**EXBI**

`public static Unit<Q> EXBI(Unit<Q> unit)`

Returns the specified unit multiplied by the factor  $2^{60}$  (binary prefix).

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1152921504606846976L).`

## Class NonSI.IndianPrefix

[javax.measure.unit](#)

```
java.lang.Object
└─ javax.measure.unit.NonSI.IndianPrefix
```

Enclosing class:

[NonSI](#)

```
public static class NonSI.IndianPrefix
extends Object
```

Inner class holding prefixes used today in India, Pakistan, Bangladesh, Nepal and Myanmar (Burma); based on grouping by two decimal places, rather than the three decimal places common in most parts of the world.

```
import static javax.measure.unit.NonSI.IndianPrefix.*; // Static import.
...
Unit<Pressure> LAKH_PASCAL = LAKH(PASCAL);
Unit<Length> CRORE_METER = CRORE(METER);
```

See Also:

[Wikipedia: Indian numbering system](#)

Method Summary		Page
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">ARAWB</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) अरब (Arawb)Returns the specified unit multiplied by the factor $10^9$	156
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">CRORE</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) करोड़ (Crore)Returns the specified unit multiplied by the factor $10^7$	156
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">DAS</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) दस (Das)Returns the specified unit multiplied by the factor $10^1$	155
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">EK</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) एक (Ek)Returns the specified unit multiplied by the factor 1	155
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">HAZAAR</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) हजार (Hazaar)Equivalent to <a href="#">SAHASR()</a> .	155
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">KHARAWB</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) खरब (Kharawb)Returns the specified unit multiplied by the factor $10^{11}$	156
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">LAKH</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) लाख (Lakh)Returns the specified unit multiplied by the factor $10^5$	156
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">MAHASHANKH</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) महाशंख (Mahashankh)Returns the specified unit multiplied by the factor $10^{19}$	157
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">NEEL</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) नील (Neel)Returns the specified unit multiplied by the factor $10^{13}$	156
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">PADMA</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) पद्म (Padma)Returns the specified unit multiplied by the factor $10^{15}$	157
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">SAHASR</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) सहस्र (Sahasr)Returns the specified unit multiplied by the factor $10^3$	155
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">SAU</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) सौ (Sau)Returns the specified unit multiplied by the factor $10^2$	155
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">SHANKH</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) शंख (Shankh)Returns the specified unit multiplied by the factor $10^{17}$	157

## Method Detail

### EK

```
public static Unit<Q> EK(Unit<Q> unit)
```

एक (Ek)Returns the specified unit multiplied by the factor 1

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1)`.

---

### DAS

```
public static Unit<Q> DAS(Unit<Q> unit)
```

दस (Das)Returns the specified unit multiplied by the factor  $10^1$

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(10)`.

---

### SAU

```
public static Unit<Q> SAU(Unit<Q> unit)
```

सौ (Sau)Returns the specified unit multiplied by the factor  $10^2$

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(100)`.

---

### SAHASR

```
public static Unit<Q> SAHASR(Unit<Q> unit)
```

सहस्र (Sahasr)Returns the specified unit multiplied by the factor  $10^3$

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1e3)`.

---

### HAZAAR

```
public static Unit<Q> HAZAAR(Unit<Q> unit)
```

हजार (Hazaar)Equivalent to [SAHASR\(\)](#).

## LAKH

```
public static Unit<Q> LAKH(Unit<Q> unit)
```

लाख (Lakh)Returns the specified unit multiplied by the factor  $10^5$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e5).

---

## CRORE

```
public static Unit<Q> CRORE(Unit<Q> unit)
```

करोड़ (Crore)Returns the specified unit multiplied by the factor  $10^7$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e7).

---

## ARAWB

```
public static Unit<Q> ARAWB(Unit<Q> unit)
```

अरब (Arawb)Returns the specified unit multiplied by the factor  $10^9$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e9).

---

## KHARAWB

```
public static Unit<Q> KHARAWB(Unit<Q> unit)
```

खरब (Kharawb)Returns the specified unit multiplied by the factor  $10^{11}$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e11).

---

## NEEL

```
public static Unit<Q> NEEL(Unit<Q> unit)
```

नील (Neel)Returns the specified unit multiplied by the factor  $10^{13}$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e13).

---

## PADMA

public static [Unit](#)<Q> **PADMA** ([Unit](#)<Q> unit)

पद्म (Padma)Returns the specified unit multiplied by the factor  $10^{15}$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e15).

---

## SHANKH

public static [Unit](#)<Q> **SHANKH** ([Unit](#)<Q> unit)

शंख (Shankh)Returns the specified unit multiplied by the factor  $10^{17}$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e17).

---

## MAHASHANKH

public static [Unit](#)<Q> **MAHASHANKH** ([Unit](#)<Q> unit)

महाशंख (Mahashankh)Returns the specified unit multiplied by the factor  $10^{19}$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e19).

## Class ProductUnit

[javax.measure.unit](#)

```
java.lang.Object
├─ javax.measure.unit.Unit<Q>
│   └─ javax.measure.unit.DerivedUnit<Q>
│       └─ javax.measure.unit.ProductUnit
```

### All Implemented Interfaces:

Serializable

```
final public class ProductUnit
extends DerivedUnit<Q>
```

This class represents units formed by the product of rational powers of existing units.

This class maintains the canonical form of this product (simplest form after factorization). For example:  
 METRE.pow(2).divide(METRE) returns METRE.

### See Also:

[Unit.times\(Unit\)](#), [Unit.divide\(Unit\)](#), [Unit.pow\(int\)](#), [Unit.root\(int\)](#)

Constructor Summary	Page
<a href="#">ProductUnit</a> ( <a href="#">Unit</a> <?> productUnit) Copy constructor (allows for parameterization of product units).	158

Method Summary	Page
boolean <a href="#">equals</a> (Object that) Indicates if the specified unit can be considered equals to the one specified.	160
<a href="#">UnitConverter</a> <a href="#">getConverterToSI</a> () Returns the converter to the standard unit.	160
<a href="#">Unit</a> <? extends <a href="#">Quantity</a> > <a href="#">getUnit</a> (int index) Returns the unit element at the specified position.	159
int <a href="#">getUnitCount</a> () Returns the number of unit elements in this product.	159
int <a href="#">getUnitPow</a> (int index) Returns the power exponent of the unit element at the specified position.	159
int <a href="#">getUnitRoot</a> (int index) Returns the root exponent of the unit element at the specified position.	159
int <a href="#">hashCode</a> () Returns the hash code for this unit.	160
<a href="#">Unit</a> <Q> <a href="#">toSI</a> () Returns the standard unit from which this unit is derived.	160

## Constructor Detail

### ProductUnit

```
public ProductUnit(Unit<?> productUnit)
```

Copy constructor (allows for parameterization of product units).

**Parameters:**

`productUnit` - the product unit source.

**Throws:**

`ClassCastException` - if the specified unit is not a product unit.

## Method Detail

### getUnitCount

```
public int getUnitCount()
```

Returns the number of unit elements in this product.

**Returns:**

the number of unit elements.

---

### getUnit

```
public Unit<? extends Quantity> getUnit(int index)
```

Returns the unit element at the specified position.

**Parameters:**

`index` - the index of the unit element to return.

**Returns:**

the unit element at the specified position.

**Throws:**

`IndexOutOfBoundsException` - if `index` is out of range (`index < 0 || index >= getUnitCount()`).

---

### getUnitPow

```
public int getUnitPow(int index)
```

Returns the power exponent of the unit element at the specified position.

**Parameters:**

`index` - the index of the unit element.

**Returns:**

the unit power exponent at the specified position.

**Throws:**

`IndexOutOfBoundsException` - if `index` is out of range (`index < 0 || index >= getUnitCount()`).

---

### getUnitRoot

```
public int getUnitRoot(int index)
```

Returns the root exponent of the unit element at the specified position.

**Parameters:**

`index` - the index of the unit element.

**Returns:**

the unit root exponent at the specified position.

**Throws:**

`IndexOutOfBoundsException` - if index is out of range (`index < 0 || index >= getUnitCount()`).

---

## equals

```
public boolean equals(Object that)
```

Indicates if the specified unit can be considered equals to the one specified.

**Overrides:**

[equals](#) in class [Unit](#)

**Parameters:**

`that` - the object to compare to.

**Returns:**

`true` if this unit is considered equal to that unit; `false` otherwise.

---

## hashCode

```
public int hashCode()
```

Returns the hash code for this unit.

**Overrides:**

[hashCode](#) in class [Unit](#)

**Returns:**

this unit hashcode value.

---

## toSI

```
public Unit<Q> toSI()
```

Returns the standard unit from which this unit is derived. The SI unit identifies the "type" of [quantity](#) for which this unit is employed. For example:

```
boolean isAngularVelocity(Unit<?> u) {
    return u.toSI().equals(RADIAN.divide(SECOND));
}
assert (REVOLUTION.divide(MINUTE).isAngularVelocity());
```

**Overrides:**

[toSI](#) in class [Unit](#)

**Returns:**

the system unit this unit is derived from.

---

## getConverterToSI

```
public final UnitConverter getConverterToSI()
```

Returns the converter to the standard unit.

---

*Note: Having the same SI unit is not sufficient to ensure that a converter exists between the two units (e.g. °C/m and K/m).*

**Overrides:**

[getConverterToSI](#) in class [Unit](#)

**Returns:**

the unit converter from this unit to its standard unit.

# Class SI

[javax.measure.unit](#)

```
java.lang.Object
├─ javax.measure.unit.SystemOfUnits
│   └─ javax.measure.unit.SI
```

```
final public class SI
extends SystemOfUnits
```

This class contains SI (Système International d'Unités) base units, and derived units.

It also defines an inner class for the 20 SI prefixes used to form decimal multiples and submultiples of SI units. For example:

```
import static javax.measure.unit.SI.*; // Static import.
import static javax.measure.unit.SI.MetricPrefix.*; // Static import.
...
Unit<Pressure> HECTOPASCAL = HECTO(PASCAL);
Unit<Length> KILOMETRE = KILO(METRE);
```

See Also:

[Wikipedia: International System of Units](#)

Nested Class Summary		Page
static class	<a href="#">SI.MetricPrefix</a> Inner class holding prefixes used by the SI system.	171

Field Summary		Page
static <a href="#">BaseUnit&lt;ElectricCurrent&gt;</a>	<a href="#">AMPERE</a> The base unit for electric current quantities (A).	164
static <a href="#">Unit&lt;MagnetomotiveForce&gt;</a>	<a href="#">AMPERE_TURN</a> The unit for magnetomotive force (At).	165
static <a href="#">AlternateUnit&lt;RadioactiveActivity&gt;</a>	<a href="#">BECQUEREL</a> The derived unit for activity of a radionuclide (Bq).	168
static <a href="#">AlternateUnit&lt;DataAmount&gt;</a>	<a href="#">BIT</a> The unit for binary information (bit).	166
static <a href="#">BaseUnit&lt;LuminousIntensity&gt;</a>	<a href="#">CANDELA</a> The base unit for luminous intensity quantities (cd).	164
static <a href="#">Unit&lt;Temperature&gt;</a>	<a href="#">CELSIUS</a> The derived unit for Celsius temperature (Cel).	168
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">CENTIMETRE</a> Equivalent to CENTI (METRE).	170
static <a href="#">AlternateUnit&lt;ElectricCharge&gt;</a>	<a href="#">COULOMB</a> The derived unit for electric charge, quantity of electricity (c).	167
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">CUBIC_METRE</a> The metric unit for volume quantities (m3).	169
static <a href="#">AlternateUnit&lt;ElectricCapacitance&gt;</a>	<a href="#">FARAD</a> The derived unit for capacitance (F).	167
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">GRAM</a> The derived unit for mass quantities (g).	165

static <a href="#">AlternateUnit&lt;RadiationDoseAbsorbed&gt;</a>	<b>GRAY</b> The derived unit for absorbed dose, specific energy (imparted), kerma ( $Gy$ ).	168
static <a href="#">AlternateUnit&lt;ElectricInductance&gt;</a>	<b>HENRY</b> The derived unit for inductance ( $H$ ).	168
static <a href="#">AlternateUnit&lt;Frequency&gt;</a>	<b>HERTZ</b> The derived unit for frequency ( $Hz$ ).	166
static <a href="#">AlternateUnit&lt;Energy&gt;</a>	<b>JOULE</b> The derived unit for energy, work, quantity of heat ( $J$ ).	166
static <a href="#">AlternateUnit&lt;CatalyticActivity&gt;</a>	<b>KATAL</b> The derived unit for catalytic activity ( $kat$ ).	169
static <a href="#">BaseUnit&lt;Temperature&gt;</a>	<b>KELVIN</b> The base unit for thermodynamic temperature quantities ( $K$ ).	164
static <a href="#">BaseUnit&lt;Mass&gt;</a>	<b>KILOGRAM</b> The base unit for mass quantities ( $kg$ ).	164
static <a href="#">Unit&lt;Length&gt;</a>	<b>KILOMETRE</b> Equivalent to KILO (METRE) .	169
static <a href="#">AlternateUnit&lt;LuminousFlux&gt;</a>	<b>LUMEN</b> The derived unit for luminous flux ( $lm$ ).	168
static <a href="#">AlternateUnit&lt;Illuminance&gt;</a>	<b>LUX</b> The derived unit for illuminance ( $lx$ ).	168
static <a href="#">BaseUnit&lt;Length&gt;</a>	<b>METRE</b> The base unit for length quantities ( $m$ ).	165
static <a href="#">Unit&lt;Velocity&gt;</a>	<b>METRES_PER_SECOND</b> The metric unit for velocity quantities ( $m/s$ ).	169
static <a href="#">Unit&lt;Acceleration&gt;</a>	<b>METRES_PER_SQUARE_SECOND</b> The metric unit for acceleration quantities ( $m/s^2$ ).	169
static <a href="#">Unit&lt;Length&gt;</a>	<b>MILLIMETRE</b> Equivalent to MILLI (METRE) .	170
static <a href="#">BaseUnit&lt;AmountOfSubstance&gt;</a>	<b>MOLE</b> The base unit for amount of substance quantities ( $mol$ ).	165
static <a href="#">AlternateUnit&lt;Force&gt;</a>	<b>NEWTON</b> The derived unit for force ( $N$ ).	166
static <a href="#">AlternateUnit&lt;ElectricResistance&gt;</a>	<b>OHM</b> The derived unit for electric resistance ( $Ohm$ ).	167
static <a href="#">AlternateUnit&lt;Pressure&gt;</a>	<b>PASCAL</b> The derived unit for pressure, stress ( $Pa$ ).	166
static <a href="#">AlternateUnit&lt;Angle&gt;</a>	<b>RADIAN</b> The unit for plane angle quantities ( $rad$ ).	165
static <a href="#">BaseUnit&lt;Duration&gt;</a>	<b>SECOND</b> The base unit for duration quantities ( $s$ ).	165
static <a href="#">AlternateUnit&lt;ElectricConductance&gt;</a>	<b>SIEMENS</b> The derived unit for electric conductance ( $S$ ).	167
static <a href="#">AlternateUnit&lt;RadiationDoseEffective&gt;</a>	<b>SIEVERT</b> The derived unit for dose equivalent ( $Sv$ ).	169
static <a href="#">Unit&lt;Area&gt;</a>	<b>SQUARE_METRE</b> The metric unit for area quantities ( $m^2$ ).	169
static <a href="#">AlternateUnit&lt;SolidAngle&gt;</a>	<b>STERADIAN</b> The unit for solid angle quantities ( $sr$ ).	166

<code>static <a href="#">AlternateUnit&lt;MagneticFluxDensity&gt;</a></code>	<a href="#">TESLA</a> The derived unit for magnetic flux density ( $\text{T}$ ).	168
<code>static <a href="#">AlternateUnit&lt;ElectricPotential&gt;</a></code>	<a href="#">VOLT</a> The derived unit for electric potential difference, electromotive force ( $\text{V}$ ).	167
<code>static <a href="#">AlternateUnit&lt;Power&gt;</a></code>	<a href="#">WATT</a> The derived unit for power, radiant, flux ( $\text{W}$ ).	166
<code>static <a href="#">AlternateUnit&lt;MagneticFlux&gt;</a></code>	<a href="#">WEBER</a> The derived unit for magnetic flux ( $\text{Wb}$ ).	167

Method Summary		Page
<code>static <a href="#">SI</a></code>	<a href="#">getInstance</a> () Returns the unique instance of this class.	170
<code>Set&lt;<a href="#">Unit&lt;?&gt;&gt;</a></code>	<a href="#">getUnits</a> () Returns a read only view over the units defined in this class.	170

## Field Detail

### AMPERE

```
public static final BaseUnit<ElectricCurrent> AMPERE
```

The base unit for electric current quantities ( $\text{A}$ ). The Ampere is that constant current which, if maintained in two straight parallel conductors of infinite length, of negligible circular cross-section, and placed 1 meter apart in vacuum, would produce between these conductors a force equal to  $2 \cdot 10^{-7}$  newton per meter of length. It is named after the French physicist Andre Ampere (1775-1836).

### CANDELA

```
public static final BaseUnit<LuminousIntensity> CANDELA
```

The base unit for luminous intensity quantities ( $\text{cd}$ ). The candela is the luminous intensity, in a given direction, of a source that emits monochromatic radiation of frequency  $540 \cdot 10^{12}$  hertz and that has a radiant intensity in that direction of  $1/683$  watt per steradian

#### See Also:

[Wikipedia: Candela](#)

### KELVIN

```
public static final BaseUnit<Temperature> KELVIN
```

The base unit for thermodynamic temperature quantities ( $\text{K}$ ). The kelvin is the  $1/273.16$ th of the thermodynamic temperature of the triple point of water. It is named after the Scottish mathematician and physicist William Thomson 1st Lord Kelvin (1824-1907)

### KILOGRAM

```
public static final BaseUnit<Mass> KILOGRAM
```

The base unit for mass quantities ( $\text{kg}$ ). It is the only SI unit with a prefix as part of its name and symbol. The kilogram is equal to the mass of an international prototype in the form of a platinum-iridium cylinder kept at Sevres in France.

**See Also:**

[GRAM](#)

---

## METRE

```
public static final BaseUnit<Length> METRE
```

The base unit for length quantities ( $\text{m}$ ). One metre was redefined in 1983 as the distance traveled by light in a vacuum in  $1/299,792,458$  of a second.

---

## MOLE

```
public static final BaseUnit<AmountOfSubstance> MOLE
```

The base unit for amount of substance quantities ( $\text{mol}$ ). The mole is the amount of substance of a system which contains as many elementary entities as there are atoms in 0.012 kilogram of carbon 12.

---

## SECOND

```
public static final BaseUnit<Duration> SECOND
```

The base unit for duration quantities ( $\text{s}$ ). It is defined as the duration of 9,192,631,770 cycles of radiation corresponding to the transition between two hyperfine levels of the ground state of cesium (1967 Standard).

---

## AMPERE\_TURN

```
public static final Unit<MagnetomotiveForce> AMPERE_TURN
```

The unit for magnetomotive force ( $\text{At}$ ).

---

## GRAM

```
public static final Unit<Mass> GRAM
```

The derived unit for mass quantities ( $\text{g}$ ). The base unit for mass quantity is [KILOGRAM](#).

---

## RADIAN

```
public static final AlternateUnit<Angle> RADIAN
```

The unit for plane angle quantities ( $\text{rad}$ ). One radian is the angle between two radii of a circle such that the length of the arc between them is equal to the radius.

---

## STERADIAN

public static final [AlternateUnit<SolidAngle>](#) **STERADIAN**

The unit for solid angle quantities ( $\text{sr}$ ). One steradian is the solid angle subtended at the center of a sphere by an area on the surface of the sphere that is equal to the radius squared. The total solid angle of a sphere is  $4\pi$  steradians.

---

## BIT

public static final [AlternateUnit<DataAmount>](#) **BIT**

The unit for binary information ( $\text{bit}$ ).

---

## HERTZ

public static final [AlternateUnit<Frequency>](#) **HERTZ**

The derived unit for frequency ( $\text{Hz}$ ). A unit of frequency equal to one cycle per second. After Heinrich Rudolf Hertz (1857-1894), German physicist who was the first to produce radio waves artificially.

---

## NEWTON

public static final [AlternateUnit<Force>](#) **NEWTON**

The derived unit for force ( $\text{N}$ ). One newton is the force required to give a mass of 1 kilogram an Force of 1 metre per second per second. It is named after the English mathematician and physicist Sir Isaac Newton (1642-1727).

---

## PASCAL

public static final [AlternateUnit<Pressure>](#) **PASCAL**

The derived unit for pressure, stress ( $\text{Pa}$ ). One pascal is equal to one newton per square meter. It is named after the French philosopher and mathematician Blaise Pascal (1623-1662).

---

## JOULE

public static final [AlternateUnit<Energy>](#) **JOULE**

The derived unit for energy, work, quantity of heat ( $\text{J}$ ). One joule is the amount of work done when an applied force of 1 newton moves through a distance of 1 metre in the direction of the force. It is named after the English physicist James Prescott Joule (1818-1889).

---

## WATT

public static final [AlternateUnit<Power>](#) **WATT**

The derived unit for power, radiant, flux ( $\text{W}$ ). One watt is equal to one joule per second. It is named after the

British scientist James Watt (1736-1819).

---

## COULOMB

```
public static final AlternateUnit<ElectricCharge> COULOMB
```

The derived unit for electric charge, quantity of electricity (C). One Coulomb is equal to the quantity of charge transferred in one second by a steady current of one ampere. It is named after the French physicist Charles Augustin de Coulomb (1736-1806).

---

## VOLT

```
public static final AlternateUnit<ElectricPotential> VOLT
```

The derived unit for electric potential difference, electromotive force (V). One Volt is equal to the difference of electric potential between two points on a conducting wire carrying a constant current of one ampere when the power dissipated between the points is one watt. It is named after the Italian physicist Count Alessandro Volta (1745-1827).

---

## FARAD

```
public static final AlternateUnit<ElectricCapacitance> FARAD
```

The derived unit for capacitance (F). One Farad is equal to the capacitance of a capacitor having an equal and opposite charge of 1 coulomb on each plate and a potential difference of 1 volt between the plates. It is named after the British physicist and chemist Michael Faraday (1791-1867).

---

## OHM

```
public static final AlternateUnit<ElectricResistance> OHM
```

The derived unit for electric resistance ( $\Omega$ ). One Ohm is equal to the resistance of a conductor in which a current of one ampere is produced by a potential of one volt across its terminals. It is named after the German physicist Georg Simon Ohm (1789-1854).

---

## SIEMENS

```
public static final AlternateUnit<ElectricConductance> SIEMENS
```

The derived unit for electric conductance (S). One Siemens is equal to one ampere per volt. It is named after the German engineer Ernst Werner von Siemens (1816-1892).

---

## WEBER

```
public static final AlternateUnit<MagneticFlux> WEBER
```

The derived unit for magnetic flux (Wb). One Weber is equal to the magnetic flux that in linking a circuit of one turn produces in it an electromotive force of one volt as it is uniformly reduced to zero within one second. It is named after the German physicist Wilhelm Eduard Weber (1804-1891).

## TESLA

public static final [AlternateUnit<MagneticFluxDensity>](#) **TESLA**

The derived unit for magnetic flux density ( $T$ ). One Tesla is equal equal to one weber per square metre. It is named after the Serbian-born American electrical engineer and physicist Nikola Tesla (1856-1943).

---

## HENRY

public static final [AlternateUnit<ElectricInductance>](#) **HENRY**

The derived unit for inductance ( $H$ ). One Henry is equal to the inductance for which an induced electromotive force of one volt is produced when the current is varied at the rate of one ampere per second. It is named after the American physicist Joseph Henry (1791-1878).

---

## CELSIUS

public static final [Unit<Temperature>](#) **CELSIUS**

The derived unit for Celsius temperature ( $^{\circ}C$ ). This is a unit of temperature such as the freezing point of water (at one atmosphere of pressure) is 0 Cel, while the boiling point is 100 Cel.

---

## LUMEN

public static final [AlternateUnit<LuminousFlux>](#) **LUMEN**

The derived unit for luminous flux ( $lm$ ). One Lumen is equal to the amount of light given out through a solid angle by a source of one candela intensity radiating equally in all directions.

---

## LUX

public static final [AlternateUnit<Illuminance>](#) **LUX**

The derived unit for illuminance ( $lx$ ). One Lux is equal to one lumen per square metre.

---

## BECQUEREL

public static final [AlternateUnit<RadioactiveActivity>](#) **BECQUEREL**

The derived unit for activity of a radionuclide ( $Bq$ ). One becquerel is the radiation caused by one disintegration per second. It is named after the French physicist, Antoine-Henri Becquerel (1852-1908).

---

## GRAY

public static final [AlternateUnit<RadiationDoseAbsorbed>](#) **GRAY**

The derived unit for absorbed dose, specific energy (imparted), kerma ( $Gy$ ). One gray is equal to the dose of one joule of energy absorbed per one kilogram of matter. It is named after the British physician L. H.

Gray (1905-1965).

---

## SIEVERT

```
public static final AlternateUnit<RadiationDoseEffective> SIEVERT
```

The derived unit for dose equivalent (sv). One Sievert is equal is equal to the actual dose, in grays, multiplied by a "quality factor" which is larger for more dangerous forms of radiation. It is named after the Swedish physicist Rolf Sievert (1898-1966).

---

## KATAL

```
public static final AlternateUnit<CatalyticActivity> KATAL
```

The derived unit for catalytic activity (kat).

---

## METRES\_PER\_SECOND

```
public static final Unit<Velocity> METRES_PER_SECOND
```

The metric unit for velocity quantities (m/s).

---

## METRES\_PER\_SQUARE\_SECOND

```
public static final Unit<Acceleration> METRES_PER_SQUARE_SECOND
```

The metric unit for acceleration quantities (m/s<sup>2</sup>).

---

## SQUARE\_METRE

```
public static final Unit<Area> SQUARE_METRE
```

The metric unit for area quantities (m<sup>2</sup>).

---

## CUBIC\_METRE

```
public static final Unit<Volume> CUBIC_METRE
```

The metric unit for volume quantities (m<sup>3</sup>).

---

## KILOMETRE

```
public static final Unit<Length> KILOMETRE
```

Equivalent to KILO (METRE).

---

## CENTIMETRE

```
public static final Unit<Length> CENTIMETRE
```

Equivalent to CENTI (METRE).

---

## MILLIMETRE

```
public static final Unit<Length> MILLIMETRE
```

Equivalent to MILLI (METRE).

## Method Detail

### getInstance

```
public static final SI getInstance()
```

Returns the unique instance of this class.

**Returns:**  
the SI instance.

---

### getUnits

```
public Set<Unit<?>> getUnits()
```

Returns a read only view over the units defined in this class.

**Overrides:**  
[getUnits](#) in class [SystemOfUnits](#)

**Returns:**  
the collection of SI units.

## Class SI.MetricPrefix

[javax.measure.unit](#)

```
java.lang.Object
└─ javax.measure.unit.SI.MetricPrefix
```

Enclosing class:

[SI](#)

```
public static class SI.MetricPrefix
extends Object
```

Inner class holding prefixes used by the SI system.

See Also:

[Wikipedia: SI Prefix](#)

Method Summary		Page
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">ATTO</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^{-18}$	175
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">CENTI</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^{-2}$	174
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">DECI</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^{-1}$	174
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">DEKA</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^1$	174
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">EXA</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^{18}$	172
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">FEMTO</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^{-15}$	175
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">GIGA</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^9$	173
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">HECTO</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^2$	174
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">KILO</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^3$	173
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">MEGA</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^6$	173
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">MICRO</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^{-6}$	175
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">MILLI</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^{-3}$	174
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">NANO</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^{-9}$	175
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">PETA</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^{15}$	172
static <a href="#">Unit&lt;Q&gt;</a>	<a href="#">PICO</a> ( <a href="#">Unit&lt;Q&gt;</a> unit) Returns the specified unit multiplied by the factor $10^{-12}$	175

static <a href="#">Unit</a> <Q>	<a href="#">TERA</a> ( <a href="#">Unit</a> <Q> unit) Returns the specified unit multiplied by the factor $10^{12}$	173
static <a href="#">Unit</a> <Q>	<a href="#">YOCTO</a> ( <a href="#">Unit</a> <Q> unit) Returns the specified unit multiplied by the factor $10^{-24}$	176
static <a href="#">Unit</a> <Q>	<a href="#">YOTTA</a> ( <a href="#">Unit</a> <Q> unit) Returns the specified unit multiplied by the factor $10^{24}$	172
static <a href="#">Unit</a> <Q>	<a href="#">ZEPTO</a> ( <a href="#">Unit</a> <Q> unit) Returns the specified unit multiplied by the factor $10^{-21}$	176
static <a href="#">Unit</a> <Q>	<a href="#">ZETTA</a> ( <a href="#">Unit</a> <Q> unit) Returns the specified unit multiplied by the factor $10^{21}$	172

## Method Detail

### YOTTA

```
public static Unit<Q> YOTTA(Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $10^{24}$

**Parameters:**

[unit](#) - any unit.

**Returns:**

[unit](#).times(1e24).

### ZETTA

```
public static Unit<Q> ZETTA(Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $10^{21}$

**Parameters:**

[unit](#) - any unit.

**Returns:**

[unit](#).times(1e21).

### EXA

```
public static Unit<Q> EXA(Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $10^{18}$

**Parameters:**

[unit](#) - any unit.

**Returns:**

[unit](#).times(1e18).

### PETA

```
public static Unit<Q> PETA(Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $10^{15}$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e15).

---

## TERA

public static [Unit](#)<Q> **TERA**([Unit](#)<Q> unit)

Returns the specified unit multiplied by the factor  $10^{12}$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e12).

---

## GIGA

public static [Unit](#)<Q> **GIGA**([Unit](#)<Q> unit)

Returns the specified unit multiplied by the factor  $10^9$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e9).

---

## MEGA

public static [Unit](#)<Q> **MEGA**([Unit](#)<Q> unit)

Returns the specified unit multiplied by the factor  $10^6$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e6).

---

## KILO

public static [Unit](#)<Q> **KILO**([Unit](#)<Q> unit)

Returns the specified unit multiplied by the factor  $10^3$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e3).

---

## HECTO

```
public static Unit<Q> HECTO (Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $10^2$

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1e2)`.

---

## DEKA

```
public static Unit<Q> DEKA (Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $10^1$

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1e1)`.

---

## DECI

```
public static Unit<Q> DECI (Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $10^{-1}$

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1e-1)`.

---

## CENTI

```
public static Unit<Q> CENTI (Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $10^{-2}$

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1e-2)`.

---

## MILLI

```
public static Unit<Q> MILLI (Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $10^{-3}$

**Parameters:**

`unit` - any unit.

**Returns:**

`unit.times(1e-3).`

---

## MICRO

public static [Unit](#)<Q> **MICRO**([Unit](#)<Q> unit)

Returns the specified unit multiplied by the factor  $10^{-6}$

**Parameters:**

unit - any unit.

**Returns:**

`unit.times(1e-6).`

---

## NANO

public static [Unit](#)<Q> **NANO**([Unit](#)<Q> unit)

Returns the specified unit multiplied by the factor  $10^{-9}$

**Parameters:**

unit - any unit.

**Returns:**

`unit.times(1e-9).`

---

## PICO

public static [Unit](#)<Q> **PICO**([Unit](#)<Q> unit)

Returns the specified unit multiplied by the factor  $10^{-12}$

**Parameters:**

unit - any unit.

**Returns:**

`unit.times(1e-12).`

---

## FEMTO

public static [Unit](#)<Q> **FEMTO**([Unit](#)<Q> unit)

Returns the specified unit multiplied by the factor  $10^{-15}$

**Parameters:**

unit - any unit.

**Returns:**

`unit.times(1e-15).`

---

## ATTO

public static final [Unit](#)<Q> **ATTO**([Unit](#)<Q> unit)

Returns the specified unit multiplied by the factor  $10^{-18}$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e-18).

---

## ZEPTO

```
public static Unit<Q> ZEPTO(Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $10^{-21}$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e-21).

---

## YOCTO

```
public static Unit<Q> YOCTO(Unit<Q> unit)
```

Returns the specified unit multiplied by the factor  $10^{-24}$

**Parameters:**

unit - any unit.

**Returns:**

unit.times(1e-24).

# Class SystemOfUnits

[javax.measure.unit](#)

```
java.lang.Object  
└─ javax.measure.unit.SystemOfUnits
```

## Direct Known Subclasses:

[NonSI](#), [SI](#), [UCUM](#)

```
abstract public class SystemOfUnits  
extends Object
```

This class represents a system of units, it groups units together for historical or cultural reasons. Nothing prevents a unit from belonging to several system of units at the same time (for example an imperial system would have many of the units held by [NonSI](#)).

Constructor Summary	Page
<a href="#">SystemOfUnits</a> ()	177

Method Summary	Page
<pre>abstract Set&lt;<a href="#">Unit</a>&lt;?&gt;&gt;</pre> <a href="#">getUnits</a> () Returns a read only view over the units defined in this system.	177

## Constructor Detail

### SystemOfUnits

```
public SystemOfUnits ()
```

## Method Detail

### getUnits

```
public abstract Set<Unit<?>> getUnits ()
```

Returns a read only view over the units defined in this system.

**Returns:**

the collection of units.

## Class TransformedUnit

[javax.measure.unit](#)

```
java.lang.Object
├── javax.measure.unit.Unit<Q>
│   └── javax.measure.unit.DerivedUnit<Q>
│       └── javax.measure.unit.TransformedUnit
```

### All Implemented Interfaces:

Serializable

```
final public class TransformedUnit
extends DerivedUnit<Q>
```

This class represents the units derived from other units using [converters](#).

Examples of transformed units:

```
CELSIUS = KELVIN.add(273.15);
FOOT = METRE.times(3048).divide(10000);
MILLISECOND = MILLI(SECOND);
```

Transformed units have no label. But like any other units, they may have labels attached to them (see [SymbolMap](#)

Instances of this class are created through the [Unit.transform\(\)](#) method.

Method Summary		Page
boolean	<a href="#">equals</a> (Object that) Indicates if the specified unit can be considered equals to the one specified.	179
<a href="#">UnitConverter</a>	<a href="#">getConverterToSI</a> () Returns the converter to the standard unit.	180
<a href="#">Unit&lt;Q&gt;</a>	<a href="#">getParentUnit</a> () Returns the parent unit for this unit.	178
int	<a href="#">hashCode</a> () Returns the hash code for this unit.	179
<a href="#">UnitConverter</a>	<a href="#">toParentUnit</a> () Returns the converter to the parent unit.	179
<a href="#">Unit&lt;Q&gt;</a>	<a href="#">toSI</a> () Returns the standard unit from which this unit is derived.	179

## Method Detail

### getParentUnit

```
public Unit<Q> getParentUnit ()
```

Returns the parent unit for this unit. The parent unit is the untransformed unit from which this unit is derived.

#### Returns:

the untransformed unit from which this unit is derived.

## toParentUnit

```
public UnitConverter toParentUnit()
```

Returns the converter to the parent unit.

**Returns:**  
the converter to the parent unit.

---

## equals

```
public boolean equals(Object that)
```

Indicates if the specified unit can be considered equals to the one specified.

**Overrides:**  
[equals](#) in class [Unit](#)

**Parameters:**  
`that` - the object to compare to.

**Returns:**  
`true` if this unit is considered equal to that unit; `false` otherwise.

---

## hashCode

```
public int hashCode()
```

Returns the hash code for this unit.

**Overrides:**  
[hashCode](#) in class [Unit](#)

**Returns:**  
this unit hashcode value.

---

## toSI

```
public Unit<Q> toSI()
```

Returns the standard unit from which this unit is derived. The SI unit identifies the "type" of [quantity](#) for which this unit is employed. For example:

```
boolean isAngularVelocity(Unit<?> u) {  
    return u.toSI().equals(RADIAN.divide(SECOND));  
}  
assert(REVOLUTION.divide(MINUTE).isAngularVelocity());
```

**Overrides:**  
[toSI](#) in class [Unit](#)

**Returns:**  
the system unit this unit is derived from.

---

## **getConverterToSI**

public [UnitConverter](#) **getConverterToSI**()

Returns the converter to the standard unit.

*Note: Having the same SI unit is not sufficient to ensure that a converter exists between the two units (e.g. °C/m and K/m).*

### **Overrides:**

[getConverterToSI](#) in class [Unit](#)

### **Returns:**

the unit converter from this unit to its standard unit.

## Class UCUM

### [javax.measure.unit](#)

`java.lang.Object`

└ [javax.measure.unit.SystemOfUnits](#)

└ `javax.measure.unit.UCUM`

```
final public class UCUM
extends SystemOfUnits
```

This class contains the units ([SI](#) and [NonSI](#)) as defined in the [Uniform Code for Units of Measure](#).

Compatibility with existing [SI](#)/[NonSI](#) units has been given priority over strict adherence to the standard. We have attempted to note every place where the definitions in this class deviate from the UCUM standard, but such notes are likely to be incomplete.

### See Also:

[UCUM](#)

Field Summary		Page
static <a href="#">Unit</a> < <a href="#">Acceleration</a> >	<a href="#">ACCELERATION_OF_FREEFALL</a> As per <a href="#">UCUM</a> standard.	200
static <a href="#">Unit</a> < <a href="#">Area</a> >	<a href="#">ACRE_BRITISH</a> As per <a href="#">UCUM</a> standard.	209
static <a href="#">Unit</a> < <a href="#">Area</a> >	<a href="#">ACRE_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	206
static <a href="#">Unit</a> < <a href="#">ElectricCurrent</a> >	<a href="#">AMPERE</a> We deviate slightly from the standard here, to maintain compatability with the existing SI units.	193
static <a href="#">Unit</a> < <a href="#">MagnetomotiveForce</a> >	<a href="#">AMPERE_TURN</a>	194
static <a href="#">Unit</a> < <a href="#">Length</a> >	<a href="#">ANGSTROM</a> As per <a href="#">UCUM</a> standard.	218
static <a href="#">Unit</a> < <a href="#">Area</a> >	<a href="#">ARE</a> As per <a href="#">UCUM</a> standard.	196
static <a href="#">Unit</a> < <a href="#">Length</a> >	<a href="#">ASTRONOMIC_UNIT</a> As per <a href="#">UCUM</a> standard.	198
static <a href="#">Unit</a> < <a href="#">Pressure</a> >	<a href="#">ATMOSPHERE</a> As per <a href="#">UCUM</a> standard.	200
static <a href="#">Unit</a> < <a href="#">Pressure</a> >	<a href="#">ATMOSPHERE_TECHNICAL</a> As per <a href="#">UCUM</a> standard.	218
static <a href="#">Unit</a> < <a href="#">Mass</a> >	<a href="#">ATOMIC_MASS_UNIT</a> As per <a href="#">UCUM</a> standard.	198
static <a href="#">Unit</a> < <a href="#">Pressure</a> >	<a href="#">BAR</a> As per <a href="#">UCUM</a> standard.	198
static <a href="#">Unit</a> < <a href="#">Area</a> >	<a href="#">BARN</a> As per <a href="#">UCUM</a> standard.	218
static <a href="#">Unit</a> < <a href="#">Volume</a> >	<a href="#">BARREL_US</a> As per <a href="#">UCUM</a> standard.	209
static <a href="#">Unit</a> < <a href="#">DataRate</a> >	<a href="#">BAUD</a> As per <a href="#">UCUM</a> standard.	219

static <a href="#">Unit&lt;RadioactiveActivity&gt;</a>	<a href="#">BECQUEREL</a> As per <a href="#">UCUM</a> standard.	195
static <a href="#">Unit&lt;Dimensionless&gt;</a>	<a href="#">BILLIONS</a> As per <a href="#">UCUM</a> standard.	191
static <a href="#">Unit&lt;ElectricCurrent&gt;</a>	<a href="#">BIOT</a> As per <a href="#">UCUM</a> standard.	201
static <a href="#">Unit&lt;DataAmount&gt;</a>	<a href="#">BIT</a> As per <a href="#">UCUM</a> standard.	219
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">BOARD_FOOT_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	204
static <a href="#">Unit&lt;Dimensionless&gt;</a>	<a href="#">BOLTZMAN</a> As per <a href="#">UCUM</a> standard.	199
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">BTU</a> As per <a href="#">UCUM</a> standard.	218
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">BTU_AT_39F</a> As per <a href="#">UCUM</a> standard.	217
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">BTU_AT_59F</a> As per <a href="#">UCUM</a> standard.	217
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">BTU_AT_60F</a> As per <a href="#">UCUM</a> standard.	217
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">BTU_INTERNATIONAL_TABLE</a> As per <a href="#">UCUM</a> standard.	217
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">BTU_MEAN</a> As per <a href="#">UCUM</a> standard.	217
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">BTU_THERMOCHEMICAL</a> As per <a href="#">UCUM</a> standard.	217
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">BUSHEL_BRITISH</a> As per <a href="#">UCUM</a> standard.	211
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">BUSHEL_US</a> As per <a href="#">UCUM</a> standard.	210
static <a href="#">Unit&lt;DataAmount&gt;</a>	<a href="#">BYTE</a> As per <a href="#">UCUM</a> standard.	219
static <a href="#">Unit&lt;Velocity&gt;</a>	<a href="#">C</a> As per <a href="#">UCUM</a> standard.	199
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">CALORIE</a> As per <a href="#">UCUM</a> standard.	216
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">CALORIE_AT_15C</a> As per <a href="#">UCUM</a> standard.	216
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">CALORIE_AT_20C</a> As per <a href="#">UCUM</a> standard.	216
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">CALORIE_FOOD</a> As per <a href="#">UCUM</a> standard.	217
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">CALORIE_INTERNATIONAL_TABLE</a> As per <a href="#">UCUM</a> standard.	216
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">CALORIE_MEAN</a> As per <a href="#">UCUM</a> standard.	216
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">CALORIE_THERMOCHEMICAL</a> As per <a href="#">UCUM</a> standard.	216
static <a href="#">BaseUnit&lt;LuminousIntensity&gt;</a>	<a href="#">CANDELA</a> As per <a href="#">UCUM</a> standard.	191

static <a href="#">Unit&lt;Dimensionless&gt;</a>	<a href="#">CARAT_GOLD</a> As per <a href="#">UCUM</a> standard.	219
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">CARAT_METRIC</a> As per <a href="#">UCUM</a> standard.	219
static <a href="#">Unit&lt;Temperature&gt;</a>	<a href="#">CELSIUS</a> As per <a href="#">UCUM</a> standard.	194
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">CHAIN_BRITISH</a> As per <a href="#">UCUM</a> standard.	208
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">CHAIN_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	206
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">CICERO</a> As per <a href="#">UCUM</a> standard.	216
static <a href="#">Unit&lt;Pressure&gt;</a>	<a href="#">CIRCLE</a> As per <a href="#">UCUM</a> standard.	219
static <a href="#">Unit&lt;Area&gt;</a>	<a href="#">CIRCULAR_MIL_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	205
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">CORD_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	204
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">CORD_US</a> As per <a href="#">UCUM</a> standard.	210
static <a href="#">AlternateUnit&lt;ElectricCharge&gt;</a>	<a href="#">COULOMB</a> As per <a href="#">UCUM</a> standard.	191
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">CUBIC_FOOT_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	204
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">CUBIC_INCH_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	204
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">CUBIC_YARD_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	204
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">CUP_US</a> As per <a href="#">UCUM</a> standard.	211
static <a href="#">Unit&lt;RadioactiveActivity&gt;</a>	<a href="#">CURIE</a> As per <a href="#">UCUM</a> standard.	202
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">DAY</a> As per <a href="#">UCUM</a> standard.	197
static <a href="#">Unit&lt;Angle&gt;</a>	<a href="#">DEGREE</a> We deviate slightly from the standard here, to maintain compatibility with the existing NonSI units.	195
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">DIDOT</a> As per <a href="#">UCUM</a> standard.	215
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">DRAM</a> As per <a href="#">UCUM</a> standard.	213
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">DRAM_APOTHECARY</a> As per <a href="#">UCUM</a> standard.	214
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">DRY_PINT_US</a> As per <a href="#">UCUM</a> standard.	210
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">DRY_QUART_US</a> As per <a href="#">UCUM</a> standard.	210
static <a href="#">Unit&lt;Force&gt;</a>	<a href="#">DYNE</a> As per <a href="#">UCUM</a> standard.	201
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">ELECTRON_MASS</a> As per <a href="#">UCUM</a> standard.	199

static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">ELECTRON_VOLT</a> As per <a href="#">UCUM</a> standard.	198
static <a href="#">Unit&lt;ElectricCharge&gt;</a>	<a href="#">ELEMENTARY_CHARGE</a> As per <a href="#">UCUM</a> standard.	199
static <a href="#">Unit&lt;Energy&gt;</a>	<a href="#">ERG</a> As per <a href="#">UCUM</a> standard.	201
static <a href="#">Unit&lt;Temperature&gt;</a>	<a href="#">FAHRENHEIT</a> As per <a href="#">UCUM</a> standard.	216
static <a href="#">Unit&lt;ElectricCapacitance&gt;</a>	<a href="#">FARAD</a> As per <a href="#">UCUM</a> standard.	194
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">FATHOM_BRITISH</a> As per <a href="#">UCUM</a> standard.	208
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">FATHOM_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	203
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">FATHOM_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	206
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">FLUID_DRAM_BRITISH</a> As per <a href="#">UCUM</a> standard.	212
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">FLUID_DRAM_US</a> As per <a href="#">UCUM</a> standard.	210
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">FLUID_OUNCE_BRITISH</a> As per <a href="#">UCUM</a> standard.	212
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">FLUID_OUNCE_US</a> As per <a href="#">UCUM</a> standard.	209
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">FOOT_BRITISH</a> As per <a href="#">UCUM</a> standard.	207
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">FOOT_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	203
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">FOOT_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	205
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">FURLONG_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	206
static <a href="#">Unit&lt;Acceleration&gt;</a>	<a href="#">GAL</a> As per <a href="#">UCUM</a> standard.	200
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">GALLON_BRITISH</a> As per <a href="#">UCUM</a> standard.	211
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">GALLON_US</a> As per <a href="#">UCUM</a> standard.	209
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">GALLON_WINCHESTER</a> As per <a href="#">UCUM</a> standard.	210
static <a href="#">Unit&lt;MagneticFluxDensity&gt;</a>	<a href="#">GAUSS</a> As per <a href="#">UCUM</a> standard.	201
static <a href="#">Unit&lt;MagnetomotiveForce&gt;</a>	<a href="#">GILBERT</a> As per <a href="#">UCUM</a> standard.	202
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">GILL_BRITISH</a> As per <a href="#">UCUM</a> standard.	212
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">GILL_US</a> As per <a href="#">UCUM</a> standard.	209
static <a href="#">Unit&lt;Angle&gt;</a>	<a href="#">GON</a> As per <a href="#">UCUM</a> standard.	196

<code>static Unit&lt;Angle&gt;</code>	<a href="#"><u>GRADE</u></a> We deviate slightly from the standard here, to maintain compatibility with the existing NonSI units.	196
<code>static Unit&lt;Mass&gt;</code>	<a href="#"><u>GRAIN</u></a> As per <a href="#"><u>UCUM</u></a> standard.	212
<code>static Unit&lt;Mass&gt;</code>	<a href="#"><u>GRAM</u></a> We deviate slightly from the standard here, to maintain compatibility with the existing SI units.	190
<code>static Unit&lt;Force&gt;</code>	<a href="#"><u>GRAM_FORCE</u></a> As per <a href="#"><u>UCUM</u></a> standard.	200
<code>static Unit&lt;RadiationDoseAbsorbed&gt;</code>	<a href="#"><u>GRAY</u></a> As per <a href="#"><u>UCUM</u></a> standard.	195
<code>static Unit&lt;Length&gt;</code>	<a href="#"><u>HAND_INTERNATIONAL</u></a> As per <a href="#"><u>UCUM</u></a> standard.	205
<code>static Unit&lt;ElectricInductance&gt;</code>	<a href="#"><u>HENRY</u></a> As per <a href="#"><u>UCUM</u></a> standard.	195
<code>static Unit&lt;Frequency&gt;</code>	<a href="#"><u>HERTZ</u></a> As per <a href="#"><u>UCUM</u></a> standard.	193
<code>static Unit&lt;Power&gt;</code>	<a href="#"><u>HORSEPOWER</u></a> As per <a href="#"><u>UCUM</u></a> standard.	218
<code>static Unit&lt;Duration&gt;</code>	<a href="#"><u>HOUR</u></a> As per <a href="#"><u>UCUM</u></a> standard.	197
<code>static Unit&lt;Dimensionless&gt;</code>	<a href="#"><u>HUNDREDS</u></a> As per <a href="#"><u>UCUM</u></a> standard.	192
<code>static Unit&lt;Length&gt;</code>	<a href="#"><u>INCH_BRITISH</u></a> As per <a href="#"><u>UCUM</u></a> standard.	207
<code>static Unit&lt;Length&gt;</code>	<a href="#"><u>INCH_INTERNATIONAL</u></a> As per <a href="#"><u>UCUM</u></a> standard.	203
<code>static Unit&lt;Length&gt;</code>	<a href="#"><u>INCH_US_SURVEY</u></a> As per <a href="#"><u>UCUM</u></a> standard.	205
<code>static Unit&lt;Energy&gt;</code>	<a href="#"><u>JOULE</u></a> As per <a href="#"><u>UCUM</u></a> standard.	193
<code>static Unit&lt;Wavenumber&gt;</code>	<a href="#"><u>KAYSER</u></a> As per <a href="#"><u>UCUM</u></a> standard.	200
<code>static BaseUnit&lt;Temperature&gt;</code>	<a href="#"><u>KELVIN</u></a> As per <a href="#"><u>UCUM</u></a> standard.	191
<code>static Unit&lt;Length&gt;</code>	<a href="#"><u>KNOT_BRITISH</u></a> As per <a href="#"><u>UCUM</u></a> standard.	209
<code>static Unit&lt;Velocity&gt;</code>	<a href="#"><u>KNOT_INTERNATIONAL</u></a> As per <a href="#"><u>UCUM</u></a> standard.	203
<code>static Unit&lt;Illuminance&gt;</code>	<a href="#"><u>LAMBERT</u></a> As per <a href="#"><u>UCUM</u></a> standard.	202
<code>static Unit&lt;Length&gt;</code>	<a href="#"><u>LIGHT_YEAR</u></a> As per <a href="#"><u>UCUM</u></a> standard.	200
<code>static Unit&lt;Length&gt;</code>	<a href="#"><u>LINE</u></a> As per <a href="#"><u>UCUM</u></a> standard.	214
<code>static Unit&lt;Length&gt;</code>	<a href="#"><u>LINGE</u></a> As per <a href="#"><u>UCUM</u></a> standard.	215
<code>static Unit&lt;Length&gt;</code>	<a href="#"><u>LINK_BRITISH</u></a> As per <a href="#"><u>UCUM</u></a> standard.	208

static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">LINK_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	206
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">LITER</a> As per <a href="#">UCUM</a> standard.	196
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">LONG_HUNDREDWEIGHT</a> As per <a href="#">UCUM</a> standard.	213
static <a href="#">Unit&lt;Mass&gt;</a>	<a href="#">LONG_TON</a> As per <a href="#">UCUM</a> standard.	213
static <a href="#">Unit&lt;LuminousFlux&gt;</a>	<a href="#">LUMEN</a> As per <a href="#">UCUM</a> standard.	195
static <a href="#">Unit&lt;Illuminance&gt;</a>	<a href="#">LUX</a> As per <a href="#">UCUM</a> standard.	195
static <a href="#">Unit&lt;MagneticFlux&gt;</a>	<a href="#">MAXWELL</a> As per <a href="#">UCUM</a> standard.	201
static <a href="#">BaseUnit&lt;Length&gt;</a>	<a href="#">METER</a> As per <a href="#">UCUM</a> standard.	190
static <a href="#">Unit&lt;ElectricConductance&gt;</a>	<a href="#">MHO</a> As per <a href="#">UCUM</a> standard.	218
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">MIL_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	205
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">MIL_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	207
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">MILE_BRITISH</a> As per <a href="#">UCUM</a> standard.	208
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">MILE_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	203
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">MILE_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	206
static <a href="#">Unit&lt;Dimensionless&gt;</a>	<a href="#">MILLIONS</a> As per <a href="#">UCUM</a> standard.	191
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">MINIM_BRITISH</a> As per <a href="#">UCUM</a> standard.	212
static <a href="#">Unit&lt;Volume&gt;</a>	<a href="#">MINIM_US</a> As per <a href="#">UCUM</a> standard.	210
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">MINUTE</a> As per <a href="#">UCUM</a> standard.	196
static <a href="#">Unit&lt;Angle&gt;</a>	<a href="#">MINUTE_ANGLE</a> As per <a href="#">UCUM</a> standard.	196
static <a href="#">Unit&lt;AmountOfSubstance&gt;</a>	<a href="#">MOLE</a> We deviate slightly from the standard here, to maintain compatibility with the existing SI units.	193
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">MONTH</a> As per <a href="#">UCUM</a> standard.	198
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">MONTH_GREGORIAN</a> As per <a href="#">UCUM</a> standard.	198
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">MONTH_JULIAN</a> As per <a href="#">UCUM</a> standard.	197
static <a href="#">Unit&lt;Duration&gt;</a>	<a href="#">MONTH_SYNODAL</a> As per <a href="#">UCUM</a> standard.	197
static <a href="#">Unit&lt;Length&gt;</a>	<a href="#">NAUTICAL_MILE_BRITISH</a> As per <a href="#">UCUM</a> standard.	208

static <code>Unit&lt;Length&gt;</code>	<a href="#"><u>NAUTICAL_MILE_INTERNATIONAL</u></a> As per <a href="#"><u>UCUM</u></a> standard.	203
static <code>Unit&lt;Force&gt;</code>	<a href="#"><u>NEWTON</u></a> As per <a href="#"><u>UCUM</u></a> standard.	193
static <code>Unit&lt;Dimensionless&gt;</code>	<a href="#"><u>NEWTON_CONSTANT_OF_GRAVITY</u></a> As per <a href="#"><u>UCUM</u></a> standard.	200
static <code>Unit&lt;MagneticFieldStrength&gt;</code>	<a href="#"><u>OERSTED</u></a> As per <a href="#"><u>UCUM</u></a> standard.	201
static <code>Unit&lt;ElectricResistance&gt;</code>	<a href="#"><u>OHM</u></a> As per <a href="#"><u>UCUM</u></a> standard.	194
static <code>Unit&lt;Mass&gt;</code>	<a href="#"><u>OUNCE</u></a> As per <a href="#"><u>UCUM</u></a> standard.	212
static <code>Unit&lt;Mass&gt;</code>	<a href="#"><u>OUNCE_APOTHECARY</u></a> As per <a href="#"><u>UCUM</u></a> standard.	214
static <code>Unit&lt;Mass&gt;</code>	<a href="#"><u>OUNCE_TROY</u></a> As per <a href="#"><u>UCUM</u></a> standard.	214
static <code>Unit&lt;Length&gt;</code>	<a href="#"><u>PACE_BRITISH</u></a> As per <a href="#"><u>UCUM</u></a> standard.	208
static <code>Unit&lt;Length&gt;</code>	<a href="#"><u>PARSEC</u></a> As per <a href="#"><u>UCUM</u></a> standard.	198
static <code>Unit&lt;Pressure&gt;</code>	<a href="#"><u>PASCAL</u></a> As per <a href="#"><u>UCUM</u></a> standard.	193
static <code>Unit&lt;Volume&gt;</code>	<a href="#"><u>PECK_BRITISH</u></a> As per <a href="#"><u>UCUM</u></a> standard.	211
static <code>Unit&lt;Volume&gt;</code>	<a href="#"><u>PECK_US</u></a> As per <a href="#"><u>UCUM</u></a> standard.	210
static <code>Unit&lt;Mass&gt;</code>	<a href="#"><u>PENNYWEIGHT_TROY</u></a> As per <a href="#"><u>UCUM</u></a> standard.	213
static <code>Unit&lt;Dimensionless&gt;</code>	<a href="#"><u>PER_BILLION</u></a> As per <a href="#"><u>UCUM</u></a> standard.	192
static <code>Unit&lt;Dimensionless&gt;</code>	<a href="#"><u>PER_MILLION</u></a> As per <a href="#"><u>UCUM</u></a> standard.	192
static <code>Unit&lt;Dimensionless&gt;</code>	<a href="#"><u>PER_THOUSAND</u></a> As per <a href="#"><u>UCUM</u></a> standard.	192
static <code>Unit&lt;Dimensionless&gt;</code>	<a href="#"><u>PER_TRILLION</u></a> As per <a href="#"><u>UCUM</u></a> standard.	192
static <code>Unit&lt;Dimensionless&gt;</code>	<a href="#"><u>PERCENT</u></a> As per <a href="#"><u>UCUM</u></a> standard.	192
static <code>Unit&lt;MagneticPermeability&gt;</code>	<a href="#"><u>PERMEABILITY_OF_VACUUM</u></a> As per <a href="#"><u>UCUM</u></a> standard.	199
static <code>Unit&lt;ElectricPermittivity&gt;</code>	<a href="#"><u>PERMITTIVITY_OF_VACUUM</u></a> As per <a href="#"><u>UCUM</u></a> standard.	199
static <code>Unit&lt;Illuminance&gt;</code>	<a href="#"><u>PHOT</u></a> As per <a href="#"><u>UCUM</u></a> standard.	202
static <code>Unit&lt;Dimensionless&gt;</code>	<a href="#"><u>PI</u></a> As per <a href="#"><u>UCUM</u></a> standard.	192
static <code>Unit&lt;Length&gt;</code>	<a href="#"><u>PICA</u></a> As per <a href="#"><u>UCUM</u></a> standard.	215
static <code>Unit&lt;Length&gt;</code>	<a href="#"><u>PICA_PRINTER</u></a> As per <a href="#"><u>UCUM</u></a> standard.	215

static <code>Unit&lt;Length&gt;</code>	<a href="#">PIED</a> As per <a href="#">UCUM</a> standard.	215
static <code>Unit&lt;Volume&gt;</code>	<a href="#">PINT_BRITISH</a> As per <a href="#">UCUM</a> standard.	212
static <code>Unit&lt;Volume&gt;</code>	<a href="#">PINT_US</a> As per <a href="#">UCUM</a> standard.	209
static <code>Unit&lt;Action&gt;</code>	<a href="#">PLANCK</a> As per <a href="#">UCUM</a> standard.	199
static <code>Unit&lt;Length&gt;</code>	<a href="#">POINT</a> As per <a href="#">UCUM</a> standard.	215
static <code>Unit&lt;Length&gt;</code>	<a href="#">POINT_PRINTER</a> As per <a href="#">UCUM</a> standard.	215
static <code>Unit&lt;DynamicViscosity&gt;</code>	<a href="#">POISE</a> As per <a href="#">UCUM</a> standard.	201
static <code>Unit&lt;Length&gt;</code>	<a href="#">POUCE</a> As per <a href="#">UCUM</a> standard.	215
static <code>Unit&lt;Mass&gt;</code>	<a href="#">POUND</a> As per <a href="#">UCUM</a> standard.	212
static <code>Unit&lt;Mass&gt;</code>	<a href="#">POUND_APOTHECARY</a> As per <a href="#">UCUM</a> standard.	214
static <code>Unit&lt;Force&gt;</code>	<a href="#">POUND_FORCE</a> As per <a href="#">UCUM</a> standard.	213
static <code>Unit&lt;Pressure&gt;</code>	<a href="#">POUND_PER_SQUARE_INCH</a> As per <a href="#">UCUM</a> standard.	218
static <code>Unit&lt;Mass&gt;</code>	<a href="#">POUND_TROY</a> As per <a href="#">UCUM</a> standard.	214
static <code>Unit&lt;Mass&gt;</code>	<a href="#">PROTON_MASS</a> As per <a href="#">UCUM</a> standard.	200
static <code>Unit&lt;Volume&gt;</code>	<a href="#">QUART_BRITISH</a> As per <a href="#">UCUM</a> standard.	211
static <code>Unit&lt;Volume&gt;</code>	<a href="#">QUART_US</a> As per <a href="#">UCUM</a> standard.	209
static <code>Unit&lt;RadiationDoseAbsorbed&gt;</code>	<a href="#">RAD</a> As per <a href="#">UCUM</a> standard.	202
static <code>AlternateUnit&lt;Angle&gt;</code>	<a href="#">RADIAN</a> As per <a href="#">UCUM</a> standard.	191
static <code>Unit&lt;Length&gt;</code>	<a href="#">RAMDEN_CHAIN_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	206
static <code>Unit&lt;Length&gt;</code>	<a href="#">RAMDEN_LINK_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	206
static <code>Unit&lt;RadiationDoseEffective&gt;</code>	<a href="#">REM</a> As per <a href="#">UCUM</a> standard.	203
static <code>Unit&lt;Length&gt;</code>	<a href="#">ROD_BRITISH</a> As per <a href="#">UCUM</a> standard.	207
static <code>Unit&lt;Length&gt;</code>	<a href="#">ROD_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	205
static <code>Unit&lt;IonizingRadiation&gt;</code>	<a href="#">ROENTGEN</a> As per <a href="#">UCUM</a> standard.	202
static <code>Unit&lt;Mass&gt;</code>	<a href="#">SCRUPLE_APOTHECARY</a> As per <a href="#">UCUM</a> standard.	214

static <a href="#">BaseUnit</a> <Duration>	<a href="#">SECOND</a> As per <a href="#">UCUM</a> standard.	190
static <a href="#">Unit</a> <Angle>	<a href="#">SECOND_ANGLE</a> As per <a href="#">UCUM</a> standard.	196
static <a href="#">Unit</a> <Area>	<a href="#">SECTION_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	207
static <a href="#">Unit</a> <Mass>	<a href="#">SHORT_HUNDREDWEIGHT</a> As per <a href="#">UCUM</a> standard.	213
static <a href="#">Unit</a> <Mass>	<a href="#">SHORT_TON</a> As per <a href="#">UCUM</a> standard.	213
static <a href="#">Unit</a> <ElectricConductance>	<a href="#">SIEMENS</a> As per <a href="#">UCUM</a> standard.	194
static <a href="#">Unit</a> <RadiationDoseEffective>	<a href="#">SIEVERT</a> As per <a href="#">UCUM</a> standard.	195
static <a href="#">Unit</a> <Pressure>	<a href="#">SPHERE</a> As per <a href="#">UCUM</a> standard.	219
static <a href="#">Unit</a> <Area>	<a href="#">SQUARE_FOOT_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	204
static <a href="#">Unit</a> <Area>	<a href="#">SQUARE_INCH_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	204
static <a href="#">Unit</a> <Area>	<a href="#">SQUARE_MILE_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	207
static <a href="#">Unit</a> <Area>	<a href="#">SQUARE_ROD_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	207
static <a href="#">Unit</a> <Area>	<a href="#">SQUARE_YARD_INTERNATIONAL</a> As per <a href="#">UCUM</a> standard.	204
static <a href="#">Unit</a> <SolidAngle>	<a href="#">STERADIAN</a> We deviate slightly from the standard here, to maintain compatibility with the existing SI units.	193
static <a href="#">Unit</a> <Volume>	<a href="#">STERE</a> As per <a href="#">UCUM</a> standard.	218
static <a href="#">Unit</a> <Luminance>	<a href="#">STILB</a> As per <a href="#">UCUM</a> standard.	202
static <a href="#">Unit</a> <KinematicViscosity>	<a href="#">STOKES</a> As per <a href="#">UCUM</a> standard.	201
static <a href="#">Unit</a> <Mass>	<a href="#">STONE</a> As per <a href="#">UCUM</a> standard.	213
static <a href="#">Unit</a> <Volume>	<a href="#">TABLESPOON_US</a> As per <a href="#">UCUM</a> standard.	211
static <a href="#">Unit</a> <Volume>	<a href="#">TEASPOON_US</a> As per <a href="#">UCUM</a> standard.	211
static <a href="#">Unit</a> <MagneticFluxDensity>	<a href="#">TESLA</a> As per <a href="#">UCUM</a> standard.	195
static <a href="#">Unit</a> <Dimensionless>	<a href="#">THOUSANDS</a> As per <a href="#">UCUM</a> standard.	191
static <a href="#">Unit</a> <Mass>	<a href="#">TONNE</a> As per <a href="#">UCUM</a> standard.	198
static <a href="#">Unit</a> <Area>	<a href="#">TOWNSHP_US_SURVEY</a> As per <a href="#">UCUM</a> standard.	207
static <a href="#">Unit</a> <Dimensionless>	<a href="#">TRILLIONS</a> As per <a href="#">UCUM</a> standard.	191

<code>static Unit&lt;ElectricPotential&gt;</code>	<b><a href="#">VOLT</a></b> We deviate slightly from the standard here, to maintain compatibility with the existing SI units.	194
<code>static Unit&lt;Power&gt;</code>	<b><a href="#">WATT</a></b> As per <a href="#">UCUM</a> standard.	193
<code>static Unit&lt;MagneticFlux&gt;</code>	<b><a href="#">WEBER</a></b> As per <a href="#">UCUM</a> standard.	194
<code>static Unit&lt;Length&gt;</code>	<b><a href="#">YARD_BRITISH</a></b> As per <a href="#">UCUM</a> standard.	208
<code>static Unit&lt;Length&gt;</code>	<b><a href="#">YARD_INTERNATIONAL</a></b> As per <a href="#">UCUM</a> standard.	203
<code>static Unit&lt;Length&gt;</code>	<b><a href="#">YARD_US_SURVEY</a></b> As per <a href="#">UCUM</a> standard.	205
<code>static Unit&lt;Duration&gt;</code>	<b><a href="#">YEAR</a></b> As per <a href="#">UCUM</a> standard.	197
<code>static Unit&lt;Duration&gt;</code>	<b><a href="#">YEAR_GREGORIAN</a></b> As per <a href="#">UCUM</a> standard.	197
<code>static Unit&lt;Duration&gt;</code>	<b><a href="#">YEAR_JULIAN</a></b> As per <a href="#">UCUM</a> standard.	197
<code>static Unit&lt;Duration&gt;</code>	<b><a href="#">YEAR_TROPICAL</a></b> As per <a href="#">UCUM</a> standard.	197

Method Summary		Page
<code>static UCUM</code>	<b><a href="#">getInstance()</a></b> Returns the unique instance of this class.	219
<code>Set&lt;Unit&lt;?&gt;&gt;</code>	<b><a href="#">getUnits()</a></b> Returns a read only view over the units defined in this class.	220

## Field Detail

### METER

```
public static final BaseUnit<Length> METER
```

As per [UCUM](#) standard.

### SECOND

```
public static final BaseUnit<Duration> SECOND
```

As per [UCUM](#) standard.

### GRAM

```
public static final Unit<Mass> GRAM
```

We deviate slightly from the standard here, to maintain compatibility with the existing SI units. In UCUM, the gram is the base unit of mass, rather than the kilogram. This doesn't have much effect on the units themselves, but it does make formatting the units a challenge.

## RADIAN

```
public static final AlternateUnit<Angle> RADIAN
```

As per [UCUM](#) standard.

---

## KELVIN

```
public static final BaseUnit<Temperature> KELVIN
```

As per [UCUM](#) standard.

---

## COULOMB

```
public static final AlternateUnit<ElectricCharge> COULOMB
```

As per [UCUM](#) standard.

---

## CANDELA

```
public static final BaseUnit<LuminousIntensity> CANDELA
```

As per [UCUM](#) standard.

---

## TRIILLIONS

```
public static final Unit<Dimensionless> TRIILLIONS
```

As per [UCUM](#) standard.

---

## BILLIONS

```
public static final Unit<Dimensionless> BILLIONS
```

As per [UCUM](#) standard.

---

## MILLIONS

```
public static final Unit<Dimensionless> MILLIONS
```

As per [UCUM](#) standard.

---

## THOUSANDS

```
public static final Unit<Dimensionless> THOUSANDS
```

As per [UCUM](#) standard.

---

## HUNDREDS

```
public static final Unit<Dimensionless> HUNDREDS
```

As per [UCUM](#) standard.

---

## PI

```
public static final Unit<Dimensionless> PI
```

As per [UCUM](#) standard.

---

## PERCENT

```
public static final Unit<Dimensionless> PERCENT
```

As per [UCUM](#) standard.

---

## PER\_THOUSAND

```
public static final Unit<Dimensionless> PER_THOUSAND
```

As per [UCUM](#) standard.

---

## PER\_MILLION

```
public static final Unit<Dimensionless> PER_MILLION
```

As per [UCUM](#) standard.

---

## PER\_BILLION

```
public static final Unit<Dimensionless> PER_BILLION
```

As per [UCUM](#) standard.

---

## PER\_TRILLION

```
public static final Unit<Dimensionless> PER_TRILLION
```

As per [UCUM](#) standard.

---

## MOLE

```
public static final Unit<AmountOfSubstance> MOLE
```

We deviate slightly from the standard here, to maintain compatibility with the existing SI units. In UCUM, the mole is no longer a base unit, but is defined as `Unit.ONE.times(6.0221367E23)`.

---

## STERADIAN

```
public static final Unit<SolidAngle> STERADIAN
```

We deviate slightly from the standard here, to maintain compatibility with the existing SI units. In UCUM, the steradian is defined as `RADIAN.pow(2)`.

---

## HERTZ

```
public static final Unit<Frequency> HERTZ
```

As per [UCUM](#) standard.

---

## NEWTON

```
public static final Unit<Force> NEWTON
```

As per [UCUM](#) standard.

---

## PASCAL

```
public static final Unit<Pressure> PASCAL
```

As per [UCUM](#) standard.

---

## JOULE

```
public static final Unit<Energy> JOULE
```

As per [UCUM](#) standard.

---

## WATT

```
public static final Unit<Power> WATT
```

As per [UCUM](#) standard.

---

## AMPERE

```
public static final Unit<ElectricCurrent> AMPERE
```

We deviate slightly from the standard here, to maintain compatibility with the existing SI units. In UCUM, the ampere is defined as `COULOMB.divide(SECOND)`.

---

## AMPERE\_TURN

```
public static final Unit<MagnetomotiveForce> AMPERE_TURN
```

---

## VOLT

```
public static final Unit<ElectricPotential> VOLT
```

We deviate slightly from the standard here, to maintain compatibility with the existing SI units. In UCUM, the volt is defined as `JOULE.divide(COULOMB)`.

---

## FARAD

```
public static final Unit<ElectricCapacitance> FARAD
```

As per [UCUM](#) standard.

---

## OHM

```
public static final Unit<ElectricResistance> OHM
```

As per [UCUM](#) standard.

---

## SIEMENS

```
public static final Unit<ElectricConductance> SIEMENS
```

As per [UCUM](#) standard.

---

## WEBER

```
public static final Unit<MagneticFlux> WEBER
```

As per [UCUM](#) standard.

---

## CELSIUS

```
public static final Unit<Temperature> CELSIUS
```

As per [UCUM](#) standard.

---

## TESLA

```
public static final Unit<MagneticFluxDensity> TESLA
```

As per [UCUM](#) standard.

---

## HENRY

```
public static final Unit<ElectricInductance> HENRY
```

As per [UCUM](#) standard.

---

## LUMEN

```
public static final Unit<LuminousFlux> LUMEN
```

As per [UCUM](#) standard.

---

## LUX

```
public static final Unit<Illuminance> LUX
```

As per [UCUM](#) standard.

---

## BECQUEREL

```
public static final Unit<RadioactiveActivity> BECQUEREL
```

As per [UCUM](#) standard.

---

## GRAY

```
public static final Unit<RadiationDoseAbsorbed> GRAY
```

As per [UCUM](#) standard.

---

## SIEVERT

```
public static final Unit<RadiationDoseEffective> SIEVERT
```

As per [UCUM](#) standard.

---

## DEGREE

```
public static final Unit<Angle> DEGREE
```

We deviate slightly from the standard here, to maintain compatibility with the existing NonSI units. In

UCUM, the degree is defined as `PI.times(RADIAN.divide(180))`.

---

## GRADE

```
public static final Unit<Angle> GRADE
```

We deviate slightly from the standard here, to maintain compatibility with the existing NonSI units. In UCUM, the grade is defined as `DEGREE.times(0.9)`.

---

## GON

```
public static final Unit<Angle> GON
```

As per [UCUM](#) standard.

---

## MINUTE\_ANGLE

```
public static final Unit<Angle> MINUTE_ANGLE
```

As per [UCUM](#) standard.

---

## SECOND\_ANGLE

```
public static final Unit<Angle> SECOND_ANGLE
```

As per [UCUM](#) standard.

---

## LITER

```
public static final Unit<Volume> LITER
```

As per [UCUM](#) standard.

---

## ARE

```
public static final Unit<Area> ARE
```

As per [UCUM](#) standard.

---

## MINUTE

```
public static final Unit<Duration> MINUTE
```

As per [UCUM](#) standard.

---

## HOURL

```
public static final Unit<Duration> HOURL
```

As per [UCUM](#) standard.

---

## DAY

```
public static final Unit<Duration> DAY
```

As per [UCUM](#) standard.

---

## YEAR\_TROPICAL

```
public static final Unit<Duration> YEAR_TROPICAL
```

As per [UCUM](#) standard.

---

## YEAR\_JULIAN

```
public static final Unit<Duration> YEAR_JULIAN
```

As per [UCUM](#) standard.

---

## YEAR\_GREGORIAN

```
public static final Unit<Duration> YEAR_GREGORIAN
```

As per [UCUM](#) standard.

---

## YEAR

```
public static final Unit<Duration> YEAR
```

As per [UCUM](#) standard.

---

## MONTH\_SYNODAL

```
public static final Unit<Duration> MONTH_SYNODAL
```

As per [UCUM](#) standard.

---

## MONTH\_JULIAN

```
public static final Unit<Duration> MONTH_JULIAN
```

As per [UCUM](#) standard.

## MONTH\_GREGORIAN

```
public static final Unit<Duration> MONTH_GREGORIAN
```

As per [UCUM](#) standard.

---

## MONTH

```
public static final Unit<Duration> MONTH
```

As per [UCUM](#) standard.

---

## TONNE

```
public static final Unit<Mass> TONNE
```

As per [UCUM](#) standard.

---

## BAR

```
public static final Unit<Pressure> BAR
```

As per [UCUM](#) standard.

---

## ATOMIC\_MASS\_UNIT

```
public static final Unit<Mass> ATOMIC_MASS_UNIT
```

As per [UCUM](#) standard.

---

## ELECTRON\_VOLT

```
public static final Unit<Energy> ELECTRON_VOLT
```

As per [UCUM](#) standard.

---

## ASTRONOMIC\_UNIT

```
public static final Unit<Length> ASTRONOMIC_UNIT
```

As per [UCUM](#) standard.

---

## PARSEC

```
public static final Unit<Length> PARSEC
```

As per [UCUM](#) standard.

---

## **C**

```
public static final Unit<Velocity> C
```

As per [UCUM](#) standard.

---

## **PLANCK**

```
public static final Unit<Action> PLANCK
```

As per [UCUM](#) standard.

---

## **BOLTZMAN**

```
public static final Unit<Dimensionless> BOLTZMAN
```

As per [UCUM](#) standard.

---

## **PERMITTIVITY\_OF\_VACUUM**

```
public static final Unit<ElectricPermittivity> PERMITTIVITY_OF_VACUUM
```

As per [UCUM](#) standard.

---

## **PERMEABILITY\_OF\_VACUUM**

```
public static final Unit<MagneticPermeability> PERMEABILITY_OF_VACUUM
```

As per [UCUM](#) standard.

---

## **ELEMENTARY\_CHARGE**

```
public static final Unit<ElectricCharge> ELEMENTARY_CHARGE
```

As per [UCUM](#) standard.

---

## **ELECTRON\_MASS**

```
public static final Unit<Mass> ELECTRON_MASS
```

As per [UCUM](#) standard.

---

## PROTON\_MASS

```
public static final Unit<Mass> PROTON_MASS
```

As per [UCUM](#) standard.

---

## NEWTON\_CONSTANT\_OF\_GRAVITY

```
public static final Unit<Dimensionless> NEWTON_CONSTANT_OF_GRAVITY
```

As per [UCUM](#) standard.

---

## ACCELERATION\_OF\_FREEFALL

```
public static final Unit<Acceleration> ACCELERATION_OF_FREEFALL
```

As per [UCUM](#) standard.

---

## ATMOSPHERE

```
public static final Unit<Pressure> ATMOSPHERE
```

As per [UCUM](#) standard.

---

## LIGHT\_YEAR

```
public static final Unit<Length> LIGHT_YEAR
```

As per [UCUM](#) standard.

---

## GRAM\_FORCE

```
public static final Unit<Force> GRAM_FORCE
```

As per [UCUM](#) standard.

---

## KAYSER

```
public static final Unit<Wavenumber> KAYSER
```

As per [UCUM](#) standard.

---

## GAL

```
public static final Unit<Acceleration> GAL
```

As per [UCUM](#) standard.

## DYNE

```
public static final Unit<Force> DYNE
```

As per [UCUM](#) standard.

---

## ERG

```
public static final Unit<Energy> ERG
```

As per [UCUM](#) standard.

---

## POISE

```
public static final Unit<DynamicViscosity> POISE
```

As per [UCUM](#) standard.

---

## BIOT

```
public static final Unit<ElectricCurrent> BIOT
```

As per [UCUM](#) standard.

---

## STOKES

```
public static final Unit<KinematicViscosity> STOKES
```

As per [UCUM](#) standard.

---

## MAXWELL

```
public static final Unit<MagneticFlux> MAXWELL
```

As per [UCUM](#) standard.

---

## GAUSS

```
public static final Unit<MagneticFluxDensity> GAUSS
```

As per [UCUM](#) standard.

---

## OERSTED

```
public static final Unit<MagneticFieldStrength> OERSTED
```

As per [UCUM](#) standard.

---

## GILBERT

```
public static final Unit<MagnetomotiveForce> GILBERT
```

As per [UCUM](#) standard.

---

## STILB

```
public static final Unit<Luminance> STILB
```

As per [UCUM](#) standard.

---

## LAMBERT

```
public static final Unit<Illuminance> LAMBERT
```

As per [UCUM](#) standard.

---

## PHOT

```
public static final Unit<Illuminance> PHOT
```

As per [UCUM](#) standard.

---

## CURIE

```
public static final Unit<RadioactiveActivity> CURIE
```

As per [UCUM](#) standard.

---

## ROENTGEN

```
public static final Unit<IonizingRadiation> ROENTGEN
```

As per [UCUM](#) standard.

---

## RAD

```
public static final Unit<RadiationDoseAbsorbed> RAD
```

As per [UCUM](#) standard.

---

## REM

```
public static final Unit<RadiationDoseEffective> REM
```

As per [UCUM](#) standard.

---

## INCH\_INTERNATIONAL

```
public static final Unit<Length> INCH_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## FOOT\_INTERNATIONAL

```
public static final Unit<Length> FOOT_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## YARD\_INTERNATIONAL

```
public static final Unit<Length> YARD_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## MILE\_INTERNATIONAL

```
public static final Unit<Length> MILE_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## FATHOM\_INTERNATIONAL

```
public static final Unit<Length> FATHOM_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## NAUTICAL\_MILE\_INTERNATIONAL

```
public static final Unit<Length> NAUTICAL_MILE_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## KNOT\_INTERNATIONAL

```
public static final Unit<Velocity> KNOT_INTERNATIONAL
```

As per [UCUM](#) standard.

## **SQUARE\_INCH\_INTERNATIONAL**

```
public static final Unit<Area> SQUARE_INCH_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## **SQUARE\_FOOT\_INTERNATIONAL**

```
public static final Unit<Area> SQUARE_FOOT_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## **SQUARE\_YARD\_INTERNATIONAL**

```
public static final Unit<Area> SQUARE_YARD_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## **CUBIC\_INCH\_INTERNATIONAL**

```
public static final Unit<Volume> CUBIC_INCH_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## **CUBIC\_FOOT\_INTERNATIONAL**

```
public static final Unit<Volume> CUBIC_FOOT_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## **CUBIC\_YARD\_INTERNATIONAL**

```
public static final Unit<Volume> CUBIC_YARD_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## **BOARD\_FOOT\_INTERNATIONAL**

```
public static final Unit<Volume> BOARD_FOOT_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## **CORD\_INTERNATIONAL**

```
public static final Unit<Volume> CORD_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## **MIL\_INTERNATIONAL**

```
public static final Unit<Length> MIL_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## **CIRCULAR\_MIL\_INTERNATIONAL**

```
public static final Unit<Area> CIRCULAR_MIL_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## **HAND\_INTERNATIONAL**

```
public static final Unit<Length> HAND_INTERNATIONAL
```

As per [UCUM](#) standard.

---

## **FOOT\_US\_SURVEY**

```
public static final Unit<Length> FOOT_US_SURVEY
```

As per [UCUM](#) standard.

---

## **YARD\_US\_SURVEY**

```
public static final Unit<Length> YARD_US_SURVEY
```

As per [UCUM](#) standard.

---

## **INCH\_US\_SURVEY**

```
public static final Unit<Length> INCH_US_SURVEY
```

As per [UCUM](#) standard.

---

## **ROD\_US\_SURVEY**

```
public static final Unit<Length> ROD_US_SURVEY
```

As per [UCUM](#) standard.

---

## **CHAIN\_US\_SURVEY**

public static final [Unit<Length>](#) CHAIN\_US\_SURVEY

As per [UCUM](#) standard.

---

## **LINK\_US\_SURVEY**

public static final [Unit<Length>](#) LINK\_US\_SURVEY

As per [UCUM](#) standard.

---

## **RAMDEN\_CHAIN\_US\_SURVEY**

public static final [Unit<Length>](#) RAMDEN\_CHAIN\_US\_SURVEY

As per [UCUM](#) standard.

---

## **RAMDEN\_LINK\_US\_SURVEY**

public static final [Unit<Length>](#) RAMDEN\_LINK\_US\_SURVEY

As per [UCUM](#) standard.

---

## **FATHOM\_US\_SURVEY**

public static final [Unit<Length>](#) FATHOM\_US\_SURVEY

As per [UCUM](#) standard.

---

## **FURLONG\_US\_SURVEY**

public static final [Unit<Length>](#) FURLONG\_US\_SURVEY

As per [UCUM](#) standard.

---

## **MILE\_US\_SURVEY**

public static final [Unit<Length>](#) MILE\_US\_SURVEY

As per [UCUM](#) standard.

---

## **ACRE\_US\_SURVEY**

public static final [Unit<Area>](#) ACRE\_US\_SURVEY

As per [UCUM](#) standard.

## SQUARE\_ROD\_US\_SURVEY

public static final [Unit<Area>](#) SQUARE\_ROD\_US\_SURVEY

As per [UCUM](#) standard.

---

## SQUARE\_MILE\_US\_SURVEY

public static final [Unit<Area>](#) SQUARE\_MILE\_US\_SURVEY

As per [UCUM](#) standard.

---

## SECTION\_US\_SURVEY

public static final [Unit<Area>](#) SECTION\_US\_SURVEY

As per [UCUM](#) standard.

---

## TOWNSHP\_US\_SURVEY

public static final [Unit<Area>](#) TOWNSHP\_US\_SURVEY

As per [UCUM](#) standard.

---

## MIL\_US\_SURVEY

public static final [Unit<Length>](#) MIL\_US\_SURVEY

As per [UCUM](#) standard.

---

## INCH\_BRITISH

public static final [Unit<Length>](#) INCH\_BRITISH

As per [UCUM](#) standard.

---

## FOOT\_BRITISH

public static final [Unit<Length>](#) FOOT\_BRITISH

As per [UCUM](#) standard.

---

## ROD\_BRITISH

public static final [Unit<Length>](#) ROD\_BRITISH

As per [UCUM](#) standard.

---

## **CHAIN\_BRITISH**

```
public static final Unit<Length> CHAIN_BRITISH
```

As per [UCUM](#) standard.

---

## **LINK\_BRITISH**

```
public static final Unit<Length> LINK_BRITISH
```

As per [UCUM](#) standard.

---

## **FATHOM\_BRITISH**

```
public static final Unit<Length> FATHOM_BRITISH
```

As per [UCUM](#) standard.

---

## **PACE\_BRITISH**

```
public static final Unit<Length> PACE_BRITISH
```

As per [UCUM](#) standard.

---

## **YARD\_BRITISH**

```
public static final Unit<Length> YARD_BRITISH
```

As per [UCUM](#) standard.

---

## **MILE\_BRITISH**

```
public static final Unit<Length> MILE_BRITISH
```

As per [UCUM](#) standard.

---

## **NAUTICAL\_MILE\_BRITISH**

```
public static final Unit<Length> NAUTICAL_MILE_BRITISH
```

As per [UCUM](#) standard.

---

## **KNOT\_BRITISH**

```
public static final Unit<Length> KNOT_BRITISH
```

As per [UCUM](#) standard.

---

## **ACRE\_BRITISH**

```
public static final Unit<Area> ACRE_BRITISH
```

As per [UCUM](#) standard.

---

## **GALLON\_US**

```
public static final Unit<Volume> GALLON_US
```

As per [UCUM](#) standard.

---

## **BARREL\_US**

```
public static final Unit<Volume> BARREL_US
```

As per [UCUM](#) standard.

---

## **QUART\_US**

```
public static final Unit<Volume> QUART_US
```

As per [UCUM](#) standard.

---

## **PINT\_US**

```
public static final Unit<Volume> PINT_US
```

As per [UCUM](#) standard.

---

## **GILL\_US**

```
public static final Unit<Volume> GILL_US
```

As per [UCUM](#) standard.

---

## **FLUID\_OUNCE\_US**

```
public static final Unit<Volume> FLUID_OUNCE_US
```

As per [UCUM](#) standard.

## FLUID\_DRAM\_US

```
public static final Unit<Volume> FLUID_DRAM_US
```

As per [UCUM](#) standard.

---

## MINIM\_US

```
public static final Unit<Volume> MINIM_US
```

As per [UCUM](#) standard.

---

## CORD\_US

```
public static final Unit<Volume> CORD_US
```

As per [UCUM](#) standard.

---

## BUSHEL\_US

```
public static final Unit<Volume> BUSHEL_US
```

As per [UCUM](#) standard.

---

## GALLON\_WINCHESTER

```
public static final Unit<Volume> GALLON_WINCHESTER
```

As per [UCUM](#) standard.

---

## PECK\_US

```
public static final Unit<Volume> PECK_US
```

As per [UCUM](#) standard.

---

## DRY\_QUART\_US

```
public static final Unit<Volume> DRY_QUART_US
```

As per [UCUM](#) standard.

---

## DRY\_PINT\_US

```
public static final Unit<Volume> DRY_PINT_US
```

As per [UCUM](#) standard.

---

## **TABLESPOON\_US**

```
public static final Unit<Volume> TABLESPOON_US
```

As per [UCUM](#) standard.

---

## **TEASPOON\_US**

```
public static final Unit<Volume> TEASPOON_US
```

As per [UCUM](#) standard.

---

## **CUP\_US**

```
public static final Unit<Volume> CUP_US
```

As per [UCUM](#) standard.

---

## **GALLON\_BRITISH**

```
public static final Unit<Volume> GALLON_BRITISH
```

As per [UCUM](#) standard.

---

## **PECK\_BRITISH**

```
public static final Unit<Volume> PECK_BRITISH
```

As per [UCUM](#) standard.

---

## **BUSHEL\_BRITISH**

```
public static final Unit<Volume> BUSHEL_BRITISH
```

As per [UCUM](#) standard.

---

## **QUART\_BRITISH**

```
public static final Unit<Volume> QUART_BRITISH
```

As per [UCUM](#) standard.

---

## PINT\_BRITISH

```
public static final Unit<Volume> PINT_BRITISH
```

As per [UCUM](#) standard.

---

## GILL\_BRITISH

```
public static final Unit<Volume> GILL_BRITISH
```

As per [UCUM](#) standard.

---

## FLUID\_OUNCE\_BRITISH

```
public static final Unit<Volume> FLUID_OUNCE_BRITISH
```

As per [UCUM](#) standard.

---

## FLUID\_DRAM\_BRITISH

```
public static final Unit<Volume> FLUID_DRAM_BRITISH
```

As per [UCUM](#) standard.

---

## MINIM\_BRITISH

```
public static final Unit<Volume> MINIM_BRITISH
```

As per [UCUM](#) standard.

---

## GRAIN

```
public static final Unit<Mass> GRAIN
```

As per [UCUM](#) standard.

---

## POUND

```
public static final Unit<Mass> POUND
```

As per [UCUM](#) standard.

---

## OUNCE

```
public static final Unit<Mass> OUNCE
```

As per [UCUM](#) standard.

## DRAM

```
public static final Unit<Mass> DRAM
```

As per [UCUM](#) standard.

---

## SHORT\_HUNDREDWEIGHT

```
public static final Unit<Mass> SHORT_HUNDREDWEIGHT
```

As per [UCUM](#) standard.

---

## LONG\_HUNDREDWEIGHT

```
public static final Unit<Mass> LONG_HUNDREDWEIGHT
```

As per [UCUM](#) standard.

---

## SHORT\_TON

```
public static final Unit<Mass> SHORT_TON
```

As per [UCUM](#) standard.

---

## LONG\_TON

```
public static final Unit<Mass> LONG_TON
```

As per [UCUM](#) standard.

---

## STONE

```
public static final Unit<Mass> STONE
```

As per [UCUM](#) standard.

---

## POUND\_FORCE

```
public static final Unit<Force> POUND_FORCE
```

As per [UCUM](#) standard.

---

## PENNYWEIGHT\_TROY

```
public static final Unit<Mass> PENNYWEIGHT_TROY
```

As per [UCUM](#) standard.

---

## **OUNCE\_TROY**

```
public static final Unit<Mass> OUNCE_TROY
```

As per [UCUM](#) standard.

---

## **POUND\_TROY**

```
public static final Unit<Mass> POUND_TROY
```

As per [UCUM](#) standard.

---

## **SCRUPLE\_APOTHECARY**

```
public static final Unit<Mass> SCRUPLE_APOTHECARY
```

As per [UCUM](#) standard.

---

## **DRAM\_APOTHECARY**

```
public static final Unit<Mass> DRAM_APOTHECARY
```

As per [UCUM](#) standard.

---

## **OUNCE\_APOTHECARY**

```
public static final Unit<Mass> OUNCE_APOTHECARY
```

As per [UCUM](#) standard.

---

## **POUND\_APOTHECARY**

```
public static final Unit<Mass> POUND_APOTHECARY
```

As per [UCUM](#) standard.

---

## **LINE**

```
public static final Unit<Length> LINE
```

As per [UCUM](#) standard.

---

## **POINT**

```
public static final Unit<Length> POINT
```

As per [UCUM](#) standard.

---

## **PICA**

```
public static final Unit<Length> PICA
```

As per [UCUM](#) standard.

---

## **POINT\_PRINTER**

```
public static final Unit<Length> POINT_PRINTER
```

As per [UCUM](#) standard.

---

## **PICA\_PRINTER**

```
public static final Unit<Length> PICA_PRINTER
```

As per [UCUM](#) standard.

---

## **PIED**

```
public static final Unit<Length> PIED
```

As per [UCUM](#) standard.

---

## **POUCE**

```
public static final Unit<Length> POUCE
```

As per [UCUM](#) standard.

---

## **LINGE**

```
public static final Unit<Length> LINGE
```

As per [UCUM](#) standard.

---

## **DIDOT**

```
public static final Unit<Length> DIDOT
```

As per [UCUM](#) standard.

## CICERO

```
public static final Unit<Length> CICERO
```

As per [UCUM](#) standard.

---

## FAHRENHEIT

```
public static final Unit<Temperature> FAHRENHEIT
```

As per [UCUM](#) standard.

---

## CALORIE\_AT\_15C

```
public static final Unit<Energy> CALORIE_AT_15C
```

As per [UCUM](#) standard.

---

## CALORIE\_AT\_20C

```
public static final Unit<Energy> CALORIE_AT_20C
```

As per [UCUM](#) standard.

---

## CALORIE\_MEAN

```
public static final Unit<Energy> CALORIE_MEAN
```

As per [UCUM](#) standard.

---

## CALORIE\_INTERNATIONAL\_TABLE

```
public static final Unit<Energy> CALORIE_INTERNATIONAL_TABLE
```

As per [UCUM](#) standard.

---

## CALORIE\_THERMOCHEMICAL

```
public static final Unit<Energy> CALORIE_THERMOCHEMICAL
```

As per [UCUM](#) standard.

---

## CALORIE

```
public static final Unit<Energy> CALORIE
```

As per [UCUM](#) standard.

---

## **CALORIE\_FOOD**

```
public static final Unit<Energy> CALORIE_FOOD
```

As per [UCUM](#) standard.

---

## **BTU\_AT\_39F**

```
public static final Unit<Energy> BTU_AT_39F
```

As per [UCUM](#) standard.

---

## **BTU\_AT\_59F**

```
public static final Unit<Energy> BTU_AT_59F
```

As per [UCUM](#) standard.

---

## **BTU\_AT\_60F**

```
public static final Unit<Energy> BTU_AT_60F
```

As per [UCUM](#) standard.

---

## **BTU\_MEAN**

```
public static final Unit<Energy> BTU_MEAN
```

As per [UCUM](#) standard.

---

## **BTU\_INTERNATIONAL\_TABLE**

```
public static final Unit<Energy> BTU_INTERNATIONAL_TABLE
```

As per [UCUM](#) standard.

---

## **BTU\_THERMOCHEMICAL**

```
public static final Unit<Energy> BTU_THERMOCHEMICAL
```

As per [UCUM](#) standard.

---

## BTU

```
public static final Unit<Energy> BTU
```

As per [UCUM](#) standard.

---

## HORSEPOWER

```
public static final Unit<Power> HORSEPOWER
```

As per [UCUM](#) standard.

---

## STERE

```
public static final Unit<Volume> STERE
```

As per [UCUM](#) standard.

---

## ANGSTROM

```
public static final Unit<Length> ANGSTROM
```

As per [UCUM](#) standard.

---

## BARN

```
public static final Unit<Area> BARN
```

As per [UCUM](#) standard.

---

## ATMOSPHERE\_TECHNICAL

```
public static final Unit<Pressure> ATMOSPHERE_TECHNICAL
```

As per [UCUM](#) standard.

---

## MHO

```
public static final Unit<ElectricConductance> MHO
```

As per [UCUM](#) standard.

---

## POUND\_PER\_SQUARE\_INCH

```
public static final Unit<Pressure> POUND_PER_SQUARE_INCH
```

As per [UCUM](#) standard.

---

## CIRCLE

```
public static final Unit<Pressure> CIRCLE
```

As per [UCUM](#) standard.

---

## SPHERE

```
public static final Unit<Pressure> SPHERE
```

As per [UCUM](#) standard.

---

## CARAT\_METRIC

```
public static final Unit<Mass> CARAT_METRIC
```

As per [UCUM](#) standard.

---

## CARAT\_GOLD

```
public static final Unit<Dimensionless> CARAT_GOLD
```

As per [UCUM](#) standard.

---

## BIT

```
public static final Unit<DataAmount> BIT
```

As per [UCUM](#) standard.

---

## BYTE

```
public static final Unit<DataAmount> BYTE
```

As per [UCUM](#) standard.

---

## BAUD

```
public static final Unit<DataRate> BAUD
```

As per [UCUM](#) standard.

---

## Method Detail

### `getInstance`

```
public static UCUM getInstance()
```

Returns the unique instance of this class.

**Returns:**  
the UCUM instance.

---

## **getUnits**

```
public Set<Unit<?>> getUnits()
```

Returns a read only view over the units defined in this class.

**Overrides:**  
[getUnits](#) in class [SystemOfUnits](#)

**Returns:**  
the collection of SI units.

# Class Unit

[javax.measure.unit](#)

```
java.lang.Object
└─ javax.measure.unit.Unit
```

## All Implemented Interfaces:

Serializable

## Direct Known Subclasses:

[BaseUnit](#), [DerivedUnit](#)

```
abstract public class Unit
extends Object
implements Serializable
```

This class represents a determinate [quantity](#) (as of length, time, heat, or value) adopted as a standard of measurement.

It is helpful to think of instances of this class as recording the history by which they are created. Thus, for example, the string "g/kg" (which is a dimensionless unit) would result from invoking the method `toString()` on a unit that was created by dividing a gram unit by a kilogram unit. Yet, "kg" divided by "kg" returns [ONE](#) and not "kg/kg" due to automatic unit factorization.

This class supports the multiplication of offsets units. The result is usually a unit not convertible to its [standard unit](#). Such units may appear in derivative quantities. For example Celsius per meter is an unit of gradient, which is common in atmospheric and oceanographic research.

Units raised at rational powers are also supported. For example the cubic root of liter is a unit compatible with meter.

Units specializations can only be defined by sub-classing either [BaseUnit](#), [DerivedUnit](#) or [AnnotatedUnit](#) (the unit constructor is package private). For example:

```
public LengthUnit extends AnnotatedUnit<Length> {
    public static LengthUnit METER = new LengthUnit(SI.METER, "myOwnUnitType");
    // Equivalent to SI.METER
    LengthUnit(Unit<Length> realUnit, String annotation) { super(realUnit, annotation); }
}
```

Instances of this class and sub-classes are immutable.

## See Also:

[Wikipedia: Units of measurement](#)

Field Summary		Page
<code>static <a href="#">Unit&lt;Dimensionless&gt;</a></code>	<a href="#">ONE</a> Holds the dimensionless unit ONE.	223
Method Summary		Page
<code><a href="#">AlternateUnit&lt;A&gt;</a></code>	<a href="#">alternate</a> (String symbol) Returns a unit equivalent to this unit but used in expressions to distinguish between quantities of a different nature but of the same dimensions.	225

<a href="#">Unit&lt;T&gt;</a>	<a href="#">asType</a> (Class<T> type) Casts this unit to a parameterized unit of specified nature or throw a <code>ClassCastException</code> if the dimension of the specified quantity and this unit's dimension do not match.	224
<a href="#">CompoundUnit&lt;Q&gt;</a>	<a href="#">compound</a> ( <a href="#">Unit&lt;Q&gt;</a> that) Returns the combination of this unit with the specified sub-unit.	226
<a href="#">Unit&lt;Q&gt;</a>	<a href="#">divide</a> (double divisor) Returns the result of dividing this unit by an approximate divisor.	228
<a href="#">Unit&lt;?&gt;</a>	<a href="#">divide</a> ( <a href="#">Unit&lt;?&gt;</a> that) Returns the quotient of this unit with the one specified.	228
<a href="#">Unit&lt;Q&gt;</a>	<a href="#">divide</a> (long divisor) Returns the result of dividing this unit by an exact divisor.	227
abstract boolean	<a href="#">equals</a> (Object that) Indicates if the specified unit can be considered equals to the one specified.	224
<a href="#">UnitConverter</a>	<a href="#">getConverterTo</a> ( <a href="#">Unit&lt;Q&gt;</a> that) Returns a converter of numeric values from this unit to another unit of same type (convenience method not raising checked exception).	225
<a href="#">UnitConverter</a>	<a href="#">getConverterToAny</a> ( <a href="#">Unit&lt;?&gt;</a> that) Returns a converter form this unit to the specified unit of type unknown.	225
abstract <a href="#">UnitConverter</a>	<a href="#">getConverterToSI</a> () Returns the converter to the standard unit.	223
<a href="#">Dimension</a>	<a href="#">getDimension</a> () Returns the dimension of this unit (depends upon the current dimensional <a href="#">model</a> ).	225
abstract int	<a href="#">hashCode</a> () Returns the hash code for this unit.	223
<a href="#">Unit&lt;?&gt;</a>	<a href="#">inverse</a> () Returns the inverse of this unit.	227
boolean	<a href="#">isCompatible</a> ( <a href="#">Unit&lt;?&gt;</a> that) Indicates if this unit is compatible with the unit specified.	224
boolean	<a href="#">isSI</a> () Indicates if this unit is a standard unit (base units and alternate units are standard units).	224
<a href="#">Unit&lt;Q&gt;</a>	<a href="#">plus</a> (double offset) Returns the result of adding an offset to this unit.	226
<a href="#">Unit&lt;?&gt;</a>	<a href="#">pow</a> (int n) Returns a unit equals to this unit raised to an exponent.	228
<a href="#">Unit&lt;?&gt;</a>	<a href="#">root</a> (int n) Returns a unit equals to the given root of this unit.	228
<a href="#">Unit&lt;Q&gt;</a>	<a href="#">times</a> (double factor) Returns the result of multiplying this unit by a an approximate factor.	227
<a href="#">Unit&lt;?&gt;</a>	<a href="#">times</a> ( <a href="#">Unit&lt;?&gt;</a> that) Returns the product of this unit with the one specified.	227
<a href="#">Unit&lt;Q&gt;</a>	<a href="#">times</a> (long factor) Returns the result of multiplying this unit by an exact factor.	227
abstract <a href="#">Unit&lt;Q&gt;</a>	<a href="#">toSI</a> () Returns the standard unit from which this unit is derived.	223
String	<a href="#">toString</a> () Returns the international <code>String</code> representation of this unit ( <a href="#">UCUM</a> based).	229
<a href="#">Unit&lt;Q&gt;</a>	<a href="#">transform</a> ( <a href="#">UnitConverter</a> operation) Returns the unit derived from this unit using the specified converter.	226

<code>static <a href="#">Unit</a>&lt;?&gt; <a href="#">valueOf</a>(CharSequence csq)</code>	Returns a unit instance that is defined from the specified character sequence using the <a href="#">standard</a> unit format ( <a href="#">UCUM</a> based).	229
---	---	-----

## Field Detail

### ONE

```
public static final Unit<Dimensionless> ONE
```

Holds the dimensionless unit ONE.

## Method Detail

### toSI

```
public abstract Unit<Q> toSI()
```

Returns the standard unit from which this unit is derived. The SI unit identifies the "type" of [quantity](#) for which this unit is employed. For example:

```
boolean isAngularVelocity(Unit<?> u) {  
    return u.toSI().equals(RADIAN.divide(SECOND));  
}  
assert (REVOLUTION.divide(MINUTE).isAngularVelocity());
```

**Returns:**  
the system unit this unit is derived from.

### getConverterToSI

```
public abstract UnitConverter getConverterToSI()
```

Returns the converter to the standard unit.

*Note: Having the same SI unit is not sufficient to ensure that a converter exists between the two units (e.g. °C/m and K/m).*

**Returns:**  
the unit converter from this unit to its standard unit.

### hashCode

```
public abstract int hashCode()
```

Returns the hash code for this unit.

**Overrides:**  
`hashCode` in class `Object`

**Returns:**  
this unit hashcode value.

## equals

```
public abstract boolean equals(Object that)
```

Indicates if the specified unit can be considered equals to the one specified.

**Overrides:**

`equals` in class `Object`

**Parameters:**

`that` - the object to compare to.

**Returns:**

`true` if this unit is considered equal to that unit; `false` otherwise.

---

## isSI

```
public boolean isSI()
```

Indicates if this unit is a standard unit (base units and alternate units are standard units). The standard unit identifies the "type" of [quantity](#) for which the unit is employed.

**Returns:**

`this.toSI().equals(this)`

---

## isCompatible

```
public final boolean isCompatible(Unit<?> that)
```

Indicates if this unit is compatible with the unit specified. Units don't need to be equals to be compatible. For example:

```
RADIAN.equals(ONE) == false  
RADIAN.isCompatible(ONE) == true
```

**Parameters:**

`that` - the other unit.

**Returns:**

`this.getDimension().equals(that.getDimension())`

**See Also:**

[getDimension\(\)](#)

---

## asType

```
public final Unit<T> asType(Class<T> type)  
    throws ClassCastException
```

Casts this unit to a parameterized unit of specified nature or throw a `ClassCastException` if the dimension of the specified quantity and this unit's dimension do not match. For example:

```
Unit<Length> LIGHT_YEAR = NonSI.C.times(NonSI.YEAR).asType(Length.class);
```

**Parameters:**

`type` - the quantity class identifying the nature of the unit.

**Returns:**

this unit parameterized with the specified type.

**Throws:**

`ClassCastException` - if the dimension of this unit is different from the specified quantity dimension.

`UnsupportedOperationException` - if the specified quantity class does not have a public static field named "UNIT" holding the SI unit for the quantity.

---

## getDimension

```
public final Dimension getDimension()
```

Returns the dimension of this unit (depends upon the current dimensional [model](#)).

**Returns:**

the dimension of this unit for the current model.

---

## getConverterTo

```
public final UnitConverter getConverterTo(Unit<Q> that)
                                     throws UnsupportedOperationException
```

Returns a converter of numeric values from this unit to another unit of same type (convenience method not raising checked exception).

**Parameters:**

`that` - the unit of same type to which to convert the numeric values.

**Returns:**

the converter from this unit to `that` unit.

**Throws:**

`UnsupportedOperationException` - if the converter cannot be constructed.

---

## getConverterToAny

```
public final UnitConverter getConverterToAny(Unit<?> that)
                                     throws ConversionException,
                                     UnsupportedOperationException
```

Returns a converter from this unit to the specified unit of type unknown. This method can be used when the dimension of the specified unit is unknown at compile-time or when the [dimensional model](#) allows for conversion between units of different type. To convert to a unit having the same parameterized type, [getConverterTo\(\[javax.measure.unit.Unit\]\(#\)\)](#) is preferred (no checked exception raised).

**Parameters:**

`that` - the unit to which to convert the numeric values.

**Returns:**

the converter from this unit to `that` unit.

**Throws:**

[ConversionException](#) - if the units are not compatible (e.g. `!this.isCompatible(that)`).

`UnsupportedOperationException` - if the converter cannot be constructed.

---

## alternate

```
public final AlternateUnit<A> alternate(String symbol)
```

Returns a unit equivalent to this unit but used in expressions to distinguish between quantities of a different nature but of the same dimensions.

Examples of alternate units:

```
Unit<Angle> RADIAN = ONE.alternate("rad");
Unit<Force> NEWTON = METRE.times(KILOGRAM).divide(SECOND.pow(2)).alternate("N");
Unit<Pressure> PASCAL = NEWTON.divide(METRE.pow(2)).alternate("Pa");
```

**Parameters:**

symbol - the new symbol for the alternate unit.

**Returns:**

the alternate unit.

**Throws:**

UnsupportedOperationException - if this unit is not a standard unit.

IllegalArgumentException - if the specified symbol is already associated to a different unit.

---

## compound

```
public final CompoundUnit<Q> compound(Unit<Q> that)
```

Returns the combination of this unit with the specified sub-unit. Compound units are typically used for formatting purpose. Examples of compound units:

```
Unit<Length> FOOT_INCH = FOOT.compound(INCH);
Unit<Duration> HOUR_MINUTE_SECOND = HOUR.compound(MINUTE).compound(SECOND);
```

**Parameters:**

that - the least significant unit to combine with this unit.

**Returns:**

the corresponding compound unit.

---

## transform

```
public final Unit<Q> transform(UnitConverter operation)
```

Returns the unit derived from this unit using the specified converter. The converter does not need to be linear. For example:

```
Unit<Dimensionless> DECIBEL = Unit.ONE.transform(
    new LogConverter(10).inverse().concatenate(
        new RationalConverter(1, 10));
```

**Parameters:**

operation - the converter from the transformed unit to this unit.

**Returns:**

the unit after the specified transformation.

---

## plus

```
public final Unit<Q> plus(double offset)
```

Returns the result of adding an offset to this unit. The returned unit is convertible with all units that are convertible with this unit.

**Parameters:**

offset - the offset added (expressed in this unit, e.g. CELSIUS = KELVIN.plus(273.15)).

**Returns:**

```
this.transform(new AddConverter(offset))
```

---

## times

```
public final Unit<Q> times(long factor)
```

Returns the result of multiplying this unit by an exact factor.

**Parameters:**

factor - the exact scale factor (e.g. KILOMETRE = METRE.times(1000)).

**Returns:**

```
this.transform(new RationalConverter(factor, 1))
```

---

## times

```
public final Unit<Q> times(double factor)
```

Returns the result of multiplying this unit by a an approximate factor.

**Parameters:**

factor - the approximate factor (e.g. ELECTRON\_MASS = KILOGRAM.times(9.10938188e-31)).

**Returns:**

```
this.transform(new MultiplyConverter(factor))
```

---

## times

```
public final Unit<?> times(Unit<?> that)
```

Returns the product of this unit with the one specified.

**Parameters:**

that - the unit multiplicand.

**Returns:**

```
this * that
```

---

## inverse

```
public final Unit<?> inverse()
```

Returns the inverse of this unit.

**Returns:**

```
1 / this
```

---

## divide

```
public final Unit<Q> divide(long divisor)
```

Returns the result of dividing this unit by an exact divisor.

**Parameters:**

divisor - the exact divisor. (e.g. QUART = GALLON\_LIQUID\_US.divide(4)).

**Returns:**

```
this.transform(new RationalConverter(1 , divisor))
```

---

## divide

```
public final Unit<Q> divide(double divisor)
```

Returns the result of dividing this unit by an approximate divisor.

**Parameters:**

divisor - the approximate divisor.

**Returns:**

```
this.transform(new MultiplyConverter(1.0 / divisor))
```

---

## divide

```
public final Unit<?> divide(Unit<?> that)
```

Returns the quotient of this unit with the one specified.

**Parameters:**

that - the unit divisor.

**Returns:**

```
this / that
```

---

## root

```
public final Unit<?> root(int n)
```

Returns a unit equals to the given root of this unit.

**Parameters:**

n - the root's order.

**Returns:**

the result of taking the given root of this unit.

**Throws:**

ArithmeticException - if n == 0.

---

## pow

```
public final Unit<?> pow(int n)
```

Returns a unit equals to this unit raised to an exponent.

**Parameters:**

n - the exponent.

**Returns:**

the result of raising this unit to the exponent.

## valueOf

```
public static Unit<?> valueOf(CharSequence csq)
```

Returns a unit instance that is defined from the specified character sequence using the [standard](#) unit format ([UCUM](#) based). This method is capable of parsing any units representations produced by [toString\(\)](#). Locale-sensitive unit formatting and parsing are handled by the [UnitFormat](#) class and its subclasses.

This method can be used to parse dimensionless units.

```
Unit<Dimensionless> PERCENT
    = Unit.valueOf("100").inverse().asType(Dimensionless.class);
```

### Parameters:

csq - the character sequence to parse.

### Returns:

```
UnitFormat.getStandard().parse(csq, new ParsePosition(0))
```

### Throws:

[IllegalArgumentException](#) - if the specified character sequence cannot be correctly parsed (e.g. not UCUM compliant).

---

## toString

```
public String toString()
```

Returns the international [String](#) representation of this unit ([UCUM](#) based). The string produced for a given unit is always the same; it is not affected by the locale. This means that it can be used as a canonical string representation for exchanging units, or as a key for a [Hashtable](#), etc. Locale-sensitive unit formatting and parsing is handled by the [UnitFormat](#) class and its subclasses.

Custom units types should override this method as they will not be recognized by the UCUM format.

### Overrides:

toString in class [Object](#)

### Returns:

```
UnitFormat.getStandard().format(this)
```

# Class `UnitFormat`

[javax.measure.unit](#)

```
java.lang.Object
├─ java.text.Format
│   └─ javax.measure.unit.UnitFormat
```

**All Implemented Interfaces:**  
Cloneable, Serializable

```
abstract public class UnitFormat
extends Format
```

This class provides the interface for formatting and parsing [units](#).

For all [SI](#) units, the 20 SI prefixes used to form decimal multiples and sub-multiples of SI units are recognized. [NonSI](#) units are directly recognized. For example:

```
Unit.valueOf("m°C").equals(SI.MILLI(SI.CELSIUS))
Unit.valueOf("kW").equals(SI.KILO(SI.WATT))
Unit.valueOf("ft").equals(SI.METRE.multiply(3048).divide(10000))
```

Constructor Summary		Page
protected	<a href="#">UnitFormat</a> () Base constructor.	230

Method Summary		Page
StringBuffer	<a href="#">format</a> (Object obj, StringBuffer toAppendTo, FieldPosition pos)	232
abstract Appendable	<a href="#">format</a> (Unit<?> unit, Appendable appendable) Formats the specified unit.	231
StringBuilder	<a href="#">format</a> (Unit<?> unit, StringBuilder dest) Convenience method equivalent to <a href="#">format(Unit, Appendable)</a> except it does not raise an IOException.	232
static <a href="#">UnitFormat</a>	<a href="#">getInstance</a> () Returns the unit format for the default locale.	231
static <a href="#">UnitFormat</a>	<a href="#">getInstance</a> (Locale locale) Returns the unit format for the specified locale.	231
static <a href="#">UnitFormat</a>	<a href="#">getStandard</a> () Returns the standard <a href="#">UCUM</a> unit format.	231
abstract <a href="#">Unit</a> <?>	<a href="#">parse</a> (CharSequence csq, ParsePosition cursor) Parses a portion of the specified <code>CharSequence</code> from the specified position to produce a unit.	232
<a href="#">Unit</a> <?>	<a href="#">parseObject</a> (String source, ParsePosition pos)	232

## Constructor Detail

### UnitFormat

```
protected UnitFormat()

    Base constructor.
```

## Method Detail

### getInstance

```
public static UnitFormat getInstance()
```

Returns the unit format for the default locale.

**Returns:**

```
LocalFormat.getInstance()
```

---

### getInstance

```
public static UnitFormat getInstance(Locale locale)
```

Returns the unit format for the specified locale.

**Parameters:**

`locale` - the locale for which the format is returned.

**Returns:**

```
LocalFormat.getInstance(java.util.Locale)
```

---

### getStandard

```
public static UnitFormat getStandard()
```

Returns the standard [UCUM](#) unit format. This is the format used by [Unit.valueOf\(CharSequence\)](#) and [Unit.toString\(\)](#).

This format uses characters range 0000-007F exclusively and **is not** locale-sensitive. For example:  
kg.m/s<sup>2</sup>

For a description of the UCUM format including BNF, see: [The Unified Code for Units of Measure](#)

**Returns:**

```
UCUMFormat.getCaseSensitiveInstance()
```

---

### format

```
public abstract Appendable format(Unit<?> unit,  
                                 Appendable appendable)  
    throws IOException
```

Formats the specified unit.

**Parameters:**

`unit` - the unit to format.

`appendable` - the appendable destination.

**Throws:**

`IOException` - if an error occurs.

---

## parse

```
public abstract Unit<?> parse(CharSequence csq,  
                               ParsePosition cursor)  
    throws IllegalArgumentException
```

Parses a portion of the specified `CharSequence` from the specified position to produce a unit. If there is no unit to parse [Unit.ONE](#) is returned.

**Parameters:**

`csq` - the `CharSequence` to parse.  
`cursor` - the cursor holding the current parsing index.

**Returns:**

the unit parsed from the specified character sub-sequence.

**Throws:**

`IllegalArgumentException` - if any problem occurs while parsing the specified character sequence (e.g. illegal syntax).

---

## format

```
public final StringBuffer format(Object obj,  
                                   StringBuffer toAppendTo,  
                                   FieldPosition pos)
```

**Overrides:**

`format` in class `Format`

---

## parseObject

```
public final Unit<?> parseObject(String source,  
                                   ParsePosition pos)
```

**Overrides:**

`parseObject` in class `Format`

---

## format

```
public final StringBuilder format(Unit<?> unit,  
                                   StringBuilder dest)
```

Convenience method equivalent to [format\(Unit, Appendable\)](#) except it does not raise an `IOException`.

**Parameters:**

`unit` - the unit to format.  
`dest` - the appendable destination.

**Returns:**

the specified `StringBuilder`.

---

## 6. 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)

## 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 as they provide an important compile time information: What kind of quantity a unit states. The standard parametrization mechanism in Java works reasonably well with quantities for the most basic operations. For example a length unit 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 Measure 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.

## 8. 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](#)
- JScience – Java Library for the Advancement of Sciences.
  - [Home Page](#)