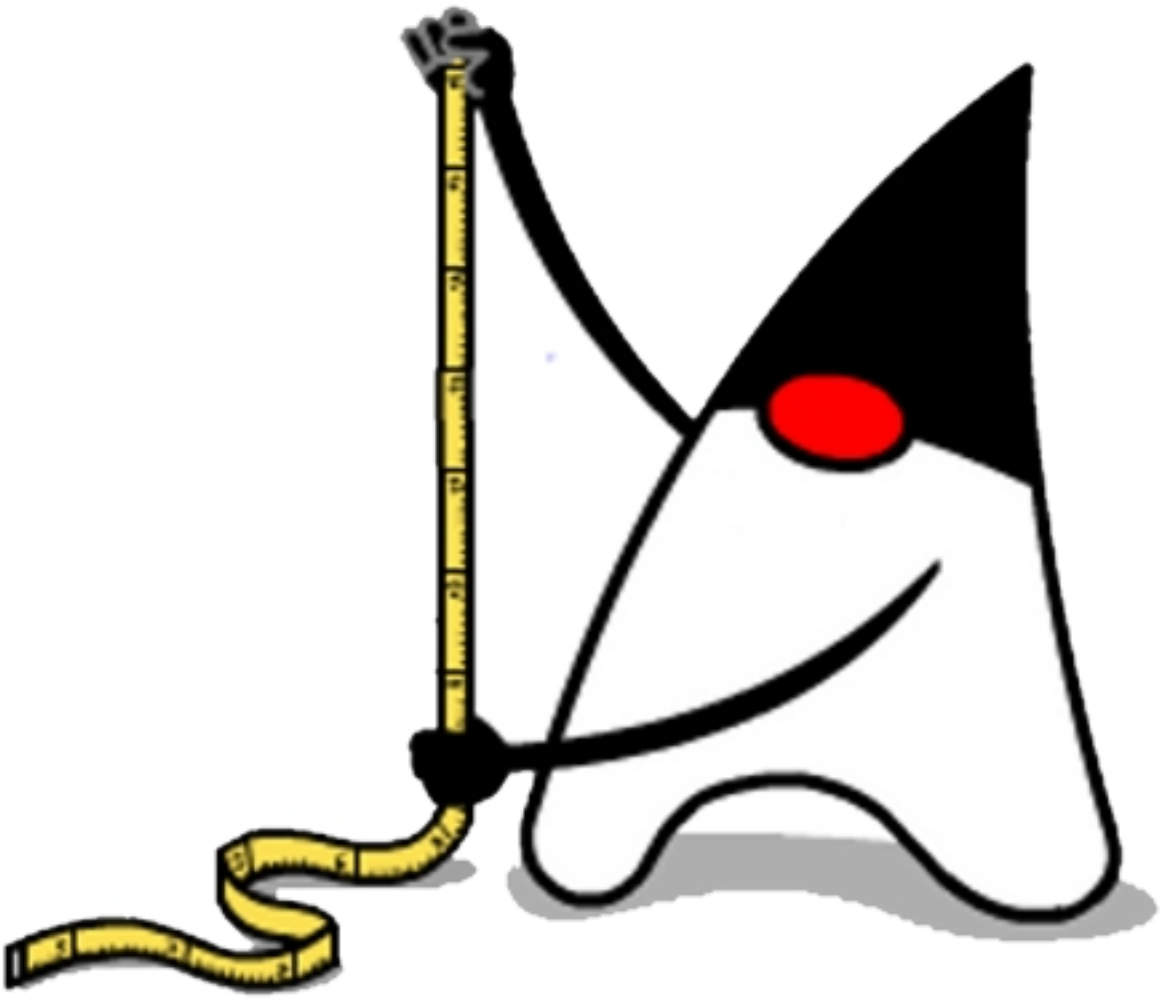
Units of Measurement Guide Book



Units of Measure

User Guide

Table of Contents

WARNING I am creating this document as I learn Indriya so it has many errors and misunderstandings. For now, use it as only an example of Asciidoctor.

Chapter 1. Introduction

1.1. Purpose

Indriya lets you write programs using dimensioned quantities instead of primitives (doubles). It is aimed at engineers, scientists and data-processors who need to write correct programs that manipulate physical units.

Advantages:

- Saves development time.
- Makes your code more readable.
- Reduce errors.
- Catches more coding errors at compile time, for example adding a length to an area.
- Makes your libraries independent of the system of units.
- You don't have to worry about what units your clients want to use, or finding conversion factors.
- More?

Indriya is the reference implementation of JSR-385 which defines a framework for handling quantities using arbitrary systems of units. However this document concentrates on using Indriya as a simple library.

JSR-385 is the latest development of the units framework started with JScience and JSR-275, so it is simple to adapt programs originally coded with JScience.

1.2. Downloading

All required jars are held in Maven Central https://mvnrepository.com/artifact.

You can get them individually, or through a Maven POM.

Versions are correct at the time of writing-check Maven for the latest version.

Artifact	Purpose	Current version	Jar
javax.measure/unit-api	Interfaces from JSR-385 (required)	2.1.2	unit-api-2.1.2.jar
tech.units/indriya	Implementation of JSR-385 (required)	2.1.1	indriya-2.1.1.jar
tech.units/uom-lib-common	Utility library (required)	2.1	uom-lib-common-2.1.jar
systems.uom/systems-common	Non-SI units (optional)	2.0.2	systems-common-2.0.2.jar

Artifact	Purpose	Current version	Jar
systems.uom/systems-quantity	Rare Non-SI units (optional)	2.0.2	systems-quantity-2.0.2.jar

1.3. Resources

The project home page is at: https://unitsofmeasurement.github.io/indriya

The Gitter page for questions is: https://gitter.im/unitsofmeasurement/indriya

The project source is in GitHub: https://github.com/unitsofmeasurement/indriya

The library Javadoc is published at: https://javadoc.io/doc/tech.units/indriya/2.1.1/tech/units/indriya/overview-summary.html

You can read the JSR at: https://jcp.org/en/jsr/detail?id=385

SI units are controlled by the BIPM. Their website is at: https://www.bipm.org/en/about-us/

Chapter 2. Getting Started with Indriya

In this chapter, we'll explain how to set up a simple Maven project and create your first quantity. Once you've done that, you're all set to explore the rest of the functionality of Indriya.

2.1. Setting Up Your Maven Project

In order to be able to use Indriya, you first have to set up your Maven project correctly. Assuming you have a working POM file, all you should need to do is to add the following dependency to your project:

```
        <dependency>
            <groupId>tech.units</groupId>
            <artifactId>indriya</artifactId>
            <version>2.1</version> <!-- 2.1.2 FAILS against si-quantity -->
            <type>jar</type>
            <scope>compile</scope>
        </dependency>
```

2.2. Your First Quantity

Next, we'll create a Java class that will create your first quantity, the speed of light (*c*). Let's start by creating a new class in a package of your choice. In that class, import the following types in order to create a new speed quantity:

```
  import javax.measure.Quantity;
  import javax.measure.quantity.Speed;
  import tech.units.indriya.quantity.Quantities;
  import tech.units.indriya.unit.Units;
```

In order to keep things simple, we'll create the quantity in the class's `main` method, and print it out immediately.

```
  public static void main(String[] args) {
      Quantity<Speed> C = Quantities.getQuantity(1079252849,
Units.KILOMETRE_PER_HOUR);
      System.out.println("The speed of light: " + C);
  }
```

Compile the class, and run it as a stand-alone Java application. The result should be like this:

```
The speed of light: 1079252849 km/h
```

That's it: you've just included Indriya as a dependency to a project, and created and printed out

your first quantity!

Chapter 3. Concepts and terms

Quantity

A quantitative property or attribute of a thing. For example, the temperature of a gas, the volume of a swimming pool, or the brightness of a star. A quantity is composed of a numeric value and the unit in which it is measured.

Quantity type

A java class representing a type of physical property without specifying which units it is measured in. For example, Length.

Unit

A measurement scale such as metre or Ampere. Each quantity type has one or more units in which it can be measured. For example a length can be measured in metres, feet or miles.

Base and derived units

In each system of units, an arbitrary set is chosen as the base units, and all the others are defined in terms of them. For example in SI the base unit for time is the second. The unit for frequency is a derived unit (1/second). This becomes important when you do arithmetic with derived units: the operations are defined on the base unit (Quirks).

Dimension

One of the fundamental aspects of quantities: length (L), mass (M), time (T), or electrical current (I). All quantity types are associated with a combination of dimensions. For example Force has dimensions of $M \cdot L \cdot T^{-2}$. Dimensions are largely hidden from the user, but they are important for conversions (see commensurable below)

Commensurable (also known as compatible)

Two units are commensurable (compatible) if they have the same dimensions. If they are commensurable, you can freely convert between them. If two quantities have commensurable units, you can compare them, and do arithmetic operations on them.

WARNING	Just because 2 quantities have commensurable units does not necessarily mean that it makes physical sense to do arithmetic with them. For example, Torque and work done both have the same dimensions so they can be added together, but the result is physically meaningless.

Is this true? needed? best here or elsewhere? Quirks?

Equivalence

Two quantities are equivalent if the physical property that they represent is the same even if the units are different. For example, 1 FOOT and 0.308 METRE are equivalent but not equal.

Chapter 4. Quantitys

4.1. Basics

A Quantity has a value, units and dimensions with getter methods for them:

```
  Quantity<Frequency> radio_one = Quantities.getQuantity(97.1,
MetricPrefix.KILO(Units.HERTZ));
  System.out.println(radio_one.getValue()); // 97100
  System.out.println(radio_one.getUnit()); // kHz
  Dimension dimension = radio_one.getUnit().getDimension();
  System.out.println(dimension); //  1/[T]
  System.out.println(dimension.getBaseDimensions()); // {[T]=-1}
```

The value is a Number, so the precision and length of the Quantity is defined by the Number class (Is this true? Expand on implications).

4.2. Creating Quantitys

For completeness, the ways to create new Quantities are:

`Quantities.getQuantity`

The normal way to create a Quantity from primitives, as illustrated above.

Arithmetic

Because Quantitys are immutable, arithmetic operations result in new Quantity objects. (Arithmetic)

Conversion

Because Quantitys are immutable, converters also result in new Quantity objects. ([sect-convertingquanties])

Parsing strings

This is normally only used when reading external input. (Parsing)

QuantityFactories

TBD (Indriya as part of JSR-365)

(De-)Serialisation

Yes (Serialisation)

4.3. Methods

Arithmetic operations - see Arithmetic

`getScale()`

Is the unit relative (like CELSIUS)? Can't think of a user application for this.

`toSystemUnit()`: Shorthand for `q.to(q.getSystemUnit())`: Can't think of a user application for this.

4.4. Utilty classes

Although they are not part of JSR-385, Indriya contains several application classes based on Quantity:

QuantityRange

Has low and high Quantitys, with a single method `contains(final Quantity<Q> q)`. See javadoc for details.

CompoundQuantity

Represents multi-radix quantities (like "1 hour, 5 min, 30 sec" or "6 ft, 3 in"). See javadoc for details.

- Should this be MixedQuantity?
- I can't parse this.

Chapter 5. Converting Quantitys

5.1. to() method

Any Quantity can be converted into an *equivalent* Quantity in different (but *commensurable*) units using the `Quantity.to()` method. It results in a new `Quantity` Object:

SI prefixes

```
Unit<Frequency> KILOHERTZ = MetricPrefix.KILO(Units.HERTZ);
Unit<Frequency> MEGAHERTZ = MetricPrefix.MEGA(Units.HERTZ);
Unit<Frequency> GIGAHERTZ = MetricPrefix.GIGA(Units.HERTZ);

Quantity<Frequency> radio_one = Quantities.getQuantity(97.1, MEGAHERTZ);
System.out.println(radio_one); // "97.1 MHz"
System.out.println(radio_one.to(KILOHERTZ)); // "97100 kHz"
System.out.println(radio_one.to(GIGAHERTZ)); // "0.0971 GHz"
```

If the units are not commensurable, you will be prevented from doing so with a compile-time error (see Error messages).

5.2. Converting primitives

If you have to convert many primitive numbers, use a `UnitConverter`:

UnitConverter

```
Unit<Length> sourceUnit = METRE;
Unit<Length> targetUnit = CENTI(METRE);
UnitConverter converter = sourceUnit.getConverterTo(targetUnit);
double length1 = 4.0;
double length2 = 6.0;
double result1 = converter.convert(length1);
double result2 = converter.convert(length2);
System.out.println(result1); // 400.0
System.out.println(result2); // 600.0
```

The (possibly complex) maths of working out the conversion factor occurs just once, and the subsequent conversions are as fast as possible.

This only converts primitives, so is a half-way house between manually controlling units and making everything a Quantity. It can be used if the performance hit of Quantity is unacceptable (Performance).

Chapter 6. Predefined Units

Normally you will use the predefined constants in the Units class explicitly: See also table at List of Predefined Units. If these are not enough, you can create derived units by various methods (Defining new Units).

Indriya does not contain Imperial or any other system of units. However they are provided by other libraries and can be freely mixed with the ones in Indriya, but this is beyond the scope of this document. (very weak-links to how?)

Chapter 7. Arithmetic

7.1. Basic operations

You can do the following arithmetic operations on Quantitys, and know that the resulting units and values are consistent:

- add (`add(Quantity)`)
- subtract (`subtract(Quantity)`)
- divide (`divide(Quantity)`)
- multiply (`multiply(Quantity)`)
- mutiply by a number (`mutiply(Number)`)
- divide by a number (`divide(Number)`)
- reciprocal (`inverse()`)
- change sign (`negate()`)

For example:

Quantity addition

```
Unit<Length> KILO_METRE = MetricPrefix.KILO(Units.METRE);
Quantity<Length> l1 = Quantities.getQuantity(200, METRE);
Quantity<Length> l2 = Quantities.getQuantity(3, KILO_METRE);
Quantity<Length> sum1 = l1.add(l2); // 3200 m
Quantity<Length> sum2 = l2.add(l1); // 3.2 km
```

Quantity multiplication

```
Unit<Length> KILO_METRE = MetricPrefix.KILO(Units.METRE);
Unit<?> WorkDone = NEWTON.multiply(METRE);
Quantity<Force> l1 = Quantities.getQuantity(200, NEWTON);
Quantity<Length> l2 = Quantities.getQuantity(3, KILO_METRE);
Quantity<?> prod1 = l1.multiply(l2); // 600 N·km
Quantity<?> prod2 = l2.multiply(l1); // 600 km·N
```

Note how the units are calculated:

- The units of the add and subtract result is always the unit of left operand of the first operation carried out.
- Similarly, the units of the multiply and divide result is assembled from the operands, in the order in which the operations are carried out.

The methods are associative and commutative and distributive just as the normal arithmetic operations are:

```
A.add(B) = B.add(A)
A.multiply(B) = B.multiply(A)
A.add(B.add(C)) = (A.add(B)).add(C)
A.multiply(B.add(C)) = (A.multiply(B)).add(A.multiply(C))
```

providing that "=" is understood as *equivalence* rather than *equality* (Concepts and terms).

The Quantity Type of an expression involving more than 1 unit cannot be determined by the compiler, hence the Quantity<?> in the multiplication example. To fix this, you can (should) specify the Quantity Type with the `.asType()` method:

```
  Quantity<Energy> prod3 = l2.multiply(l1).asType(Energy.class).to(JOULE); // 600000 J
```

Firstly it is a hint to the compiler so that it can assign the correct type. Secondly it throws a `ClassCastException` at run time if the unit does not have the same dimensions as the argument, helping to detect more errors. The method can be applied on a Unit or Quantity instance.

The unit is guaranteed to have the correct dimensions, but may not be in the form you want. The to() method converts to your preferred Unit.

7.2. Numerical errors

Divide by 0 and invalid numbers throw an `IllegalArgumentException`:

Arithmetic errors throw IllegalArgumentException

```
  getQuantity(0.0, WATT).inverse();
  getQuantity(0, WATT).inverse();
  getQuantity(10, WATT).divide(getQuantity(0, WATT));
  getQuantity(10, WATT).divide(0);
  getQuantity(Double.NaN, WATT);
  getQuantity(Double.POSITIVE_INFINITY, WATT);
```

Quantity can hold numbers larger than `Double.MAX_VALUE` (how large?), but if you try to return them as double you get INF:

Arithmetic overflow

```
  Quantity q = getQuantity(Double.MAX_VALUE,
WATT).multiply(getQuantity(Double.MAX_VALUE, WATT));
  double d = q.getValue().doubleValue();
  assertTrue(Double.isInfinite(d));
```

7.3. Quirks

The methods are mathematically correct, but this can lead to surprising results, particularly with

relative units (issue #17). For example, you might expect 2°C + 3°C to be 5°C, but when you try it the result is 278.15°C:

```
  System.out.println(getQuantity(2, CELSIUS).add(getQuantity(3, CELSIUS))); // 278.15
```

This is because the quantities are (conceptually) converted to base units first, then added, then converted back to the input units:

```
2°C + 3°C = 275.15 K + 276.15 K = 557.3 K = 278.15°C
```

Rather than adding, you probably intended to **increment** 2°C by a further 3°C. Indriya does not provide an increment method, but instead, you could use:

```
  System.out.println(getQuantity(2, CELSIUS).add(getQuantity(3, KELVIN))); // 5
```

Chapter 8. Defining new Units

There are several methods for creating derived units, each with its own uses and advantages:

- SI prefixes
- Scaling a single Unit
- Combining multiple units
- Implicitly
- Parsing

All user-defined units have exactly the same abilities as the built-in units. for example, after you have defined a unit, you can add it to the parser.

- Prove it!

8.1. SI Prefixes

Only the un-prefixed units are provided in the library. Create others as you need them:

```
  Unit<Frequency> KILOHERTZ = MetricPrefix.KILO(Units.HERTZ);
  Unit<Frequency> MEGAHERTZ = MetricPrefix.MEGA(Units.HERTZ);
  Unit<Frequency> GIGAHERTZ = MetricPrefix.GIGA(Units.HERTZ);

  Quantity<Frequency> radio_one = Quantities.getQuantity(97.1, MEGAHERTZ);
  System.out.println(radio_one); // "97.1 MHz"
  System.out.println(radio_one.to(KILOHERTZ)); // "97100 kHz"
  System.out.println(radio_one.to(GIGAHERTZ)); // "0.0971 GHz"
```

Or if you only want them occasionally, use them anonymously (but see Performance):

```
  Quantity<Frequency> radio_one = Quantities.getQuantity(97.1,
MetricPrefix.KILO(Units.HERTZ));
  System.out.println(radio_one); // "97100 kHz"
```

8.2. Scaling from existing Units

AlternateUnit makes a new unit by applying linear scaling to an existing one. The new unit has the same dimensions, but a different symbol and scale.

```
  Unit<Pressure> PASCAL = NEWTON.divide(METRE.pow(2))
          .alternate("Pa").asType(Pressure.class);
  assert SimpleUnitFormat.getInstance().parse("Pa").equals(PASCAL);
  Unit<Pressure> PASCAL = new
AlternateUnit<Pressure>(Units.NEWTON.divide(Units.METRE.pow(2)), "Pa");
```

8.3. Combining multiple Units

ProductUnit makes a new unit created as the product of rational powers of other units. The new unit has dimensions calculated from the combination of the other units.

```
  Unit<Area> squareMetre = METRE.multiply(METRE).asType(Area.class);
  Quantity<Length> line = Quantities.getQuantity(2, METRE);
  System.out.println(line.multiply(line).getUnit() == squareMetre); // true
```

8.4. Implicitly

You can use the fact that arithmetic operations manage units for you:

```

```

- Is there a use for this?

8.5. Parsing

TBD

8.6. Defining from scratch?

8.7. Register the new unit with the parser

When you have defined your new unit, let the parser know about it by:

Registering a unit symbol

```
  TBD
```

Chapter 9. Quantity types

9.1. Basics

Quantity type is just a marker interface for any property that you want:

```
```

???????

- Dont think this is true, but I don't understand it yet.
- This passes and shouldn't:

```
Quantity<Bmi> bmi5 = Quantities.getQuantity(50, KILOGRAM).asType(Bmi.class);
System.out.println(bmi5); // 50 kg !???
```

-But this shows that asType does do something with the predefined types:

```
try{
Quantity<Length> xordinate = Quantities.getQuantity(3000, WATT).asType(Length.class);
}
catch (java.lang.ClassCastException e)
{
    System.out.println(e);  // java.lang.ClassCastException: The unit: W is not
compatible with quantities of type interface javax.measure.quantity.Length
}
```

- Even if I make Bmi implement instead of extend, none of its methods get called. Where am I gouing wrong?

In Indriya there is a 1-to-many relationship between Quantity type and units (List of Predefined Units) but this is not required and is not enforced.

- Sounds wrong. Further research required

9.2. Custom Quantity types

A Quantity must have a Quantity type if you want to use it in add or subtract arithmetic. One use of custom quantity types is to label otherwise anonymous Quantitys so that you can do arithmetic with them:

```
```

It can also be used to distinguish properties that should not be mixed, but that unfortunately have the same dimensions:

```
  public interface Xordinate extends Quantity<Xordinate> {
  }

  public interface Yordinate extends Quantity<Yordinate> {
  }

  public void qtytypetest() {
      final Quantity<Length> xordinate = Quantities.getQuantity(3000, METRE);
      final Quantity<Length> yordinate = Quantities.getQuantity(500, METRE);
      // Works but not physically meaningful:
      xordinate.add(yordinate);

      final Quantity<Xordinate> xordinate2 = Quantities.getQuantity(3000,
METRE).asType(Xordinate.class);
      final Quantity<Yordinate> yordinate2 = Quantities.getQuantity(500,
METRE).asType(Yordinate.class);
      //  incompatible type compiler message:
      // xordinate2.add(yordinate2);

  }
```

NOTE This violates the 1-to-many relationship above; should it be discouraged? Is there a better way?

Chapter 10. Putting it together-a new unit of measure

This example shows how to define a new measurement unit, BMI. (BMI or body-mass-index is a widely used health statistic for measuring obesity).

```
  import static javax.measure.MetricPrefix.KILO;
  import javax.measure.Quantity;
  import javax.measure.Unit;
  import javax.measure.quantity.Length;
  import javax.measure.quantity.Mass;
  import tech.units.indriya.ComparableQuantity;
  import tech.units.indriya.format.SimpleUnitFormat;
  import tech.units.indriya.quantity.Quantities;
  import tech.units.indriya.unit.AlternateUnit;
  import tech.units.indriya.unit.Units;
  import static tech.units.indriya.unit.Units.GRAM;
  import static tech.units.indriya.unit.Units.KILOGRAM;
  import static tech.units.indriya.unit.Units.METRE;

  /**
   * Measure of human body shape. 18.5 to 24.9 is considered ideal for health.
   * Over 30 is obese.
   */
  public class BodyMassIndex {

      /**
       * Define a new Quantity type for the Bmi_unit
       */
      public interface Bmi extends Quantity<Bmi> {
      }
      /**
       * Unit to represent BMI, and attach it to the Quantity type
       */
      public static final Unit<Bmi> bmi_unit
              = new AlternateUnit<Bmi>(Units.KILOGRAM.divide(Units.METRE.pow(2)),
"BMI");

      // Register the symbol with the parser so that it knows how to parse string form
      static {
          SimpleUnitFormat.getInstance().label(bmi_unit, "BMI");
      }

      /**
       * Some useful constants
       */
      public static final Quantity<Bmi> UNDERWEIGHT
              = Quantities.getQuantity(18.5, bmi_unit);
```

```
    public static final Quantity<Bmi> OVERWEIGHT
            = Quantities.getQuantity(24.9, bmi_unit);

    public static final Quantity<Bmi> OBESE
            = Quantities.getQuantity(30.0, bmi_unit);

    /**
     * Utility method to create one. Note the return type is ComparableQuantity
     * because I expect it to be used for comparisons.
     *
     * How can I get rid of the cast?
     */
    public static ComparableQuantity<Bmi> bmi(Quantity<Mass> mass,
            Quantity<Length> height) {

        return (ComparableQuantity) mass.divide(height).divide(height)
                .asType(Bmi.class).to(bmi_unit);
    }

    /**
     * Test it
     */
    public static void main(String[] args) {

        // 1. Create one directly from measurements
        Quantity<Mass> mass = Quantities.getQuantity("75 kg").asType(Mass.class);
        Quantity<Length> height
                = Quantities.getQuantity("1.80 m").asType(Length.class);

        Quantity<Bmi> bmi1 = mass.divide(height).divide(height).asType(Bmi.class);
        System.out.println(bmi1); // 23.148148148148148148148148148148148 BMI

        // 2. Create one from the utility method
        Quantity<Bmi> bmi2 = bmi(mass, height);
        System.out.println(bmi2); // 23.148148148148148148148148148148148 BMI

        // 3. Show that the utility method accepts any commensurable units
        Quantity<Bmi> bmi3 = bmi(mass.to(GRAM), height.to(KILO(METRE)));
        System.out.println(bmi3); // 23.148148148148148148148148148148148 BMI

        // 4. Create one by parsing
        Quantity<Bmi> bmi4 = Quantities.getQuantity("27.6 BMI").asType(Bmi.class);
        System.out.println(bmi4); // 27.6 BMI

        // 5. Show that the quantity type works to prevent mismatches
        Quantity<Bmi> bmi5 = null;
        try {
            bmi5 = Quantities.getQuantity(50, KILOGRAM).asType(Bmi.class);
            System.out.println("WRONG " + bmi5); // 50 kg !???
        } catch (java.lang.ClassCastException ex) {
            // Expect java.lang.ClassCastException: The unit: kg/m² is not
```

```
compatible with quantities of type interface javax.measure.quantity.Mass
            System.out.println(ex);
        }
        // but this corrrectly spots the error
        try {
            bmi5.to(bmi_unit);
        } catch (javax.measure.UnconvertibleException ex) {
        }

        // 6. Use it in an application
        checkMyBmi(mass, height);
        // Prints "Your BMI is 23.148... BMI: You are a healthy weight"
    }

    /**
     * Sample method using Bmi.
     */
    public static void checkMyBmi(Quantity<Mass> mass, Quantity<Length> height) {

        ComparableQuantity<Bmi> bmi = bmi(mass, height);

        System.out.print("Your BMI is " + bmi + ": ");

        if (bmi.compareTo(OBESE) > 0) {
            System.out.println("You are obese");
        } else if (bmi.compareTo(OVERWEIGHT) > 0) {
            System.out.println("You are overweight");
        } else if (bmi.compareTo(UNDERWEIGHT) < 0) {
            System.out.println("You are underweight");
        } else {
            System.out.println("You are a healthy weight");
        }
    }
}
```

Chapter 11. Parsing and formatting

In JSR-265, the same class is responsible for both parsing and formatting, to maintain consistency. But it is convenient to describe parsing and formatting separately.

`UnitFormat` classes are responsible for converting units to and from String forms.

`QuantityFormat` classes do the same for Quantitys.

`SimpleUnitFormat` and `SimpleQuantityFormat` are the easiest to access and most complete, so will be described here. These provide formatters for SI units only.

For a description of the other formatters, see [appendix-spi].

11.1. Parsing

`SimpleQuantityFormat` can interpret decimal numbers, rational numbers, unit symbols, compound units and SI prefixes.[1] (`SimpleQuantityFormat` contains a `SimpleUnitFormat` so this description covers `SimpleUnitFormat` as well).

Typical usage is:

```
SimpleQuantityFormat f = SimpleQuantityFormat.getInstance();
String output = f.parse(input).toString();
```

Table 1. Parsing examples

Case	Input (Note 1,2,3,4)	Output (Note 5)
Simple	[6 m]	6 m
Extra space	[6 m]	6 m
Prefix	[11.3 kV]	11.3 kV
Rational number	[-5÷3 m]	-1.66666666666666666666666666666667 m
Compound	[11 N·m]	11 N·m
Inverted	[6 m/s]	6 m/s
Inverted powers	[6 m/s²]	6 m/s²
Repeated units	[1013 kg/m·m·m]	1013 kg/m³
Scientific	[6 m·s□²]	javax.measure.format.MeasurementParseException: Parse Error
Leading 1	[6 1/s]	javax.measure.format.MeasurementParseException: unexpected token INTEGER

Case	Input (Note 1,2,3,4)	Output (Note 5)
Alias (Note 6)	27.6 R	27.6 R
Custom (Note 7)	27.6 BMI	27.6 BMI

Notes

1. The prefix has no separator between it and the unit.
2. There must be a Unicode middle dot (not dot, full space or small space) between compound units. (Unicode characters for unit symbols)
3. Extra whitespace between the number and the unit is ignored.
4. To use numerals to specify powers, use the unicode superscript digits. (Unicode characters for unit symbols)
5. In all cases, the output can be toString()ed and reparsed, and the result is equal and equivalent to the first parse result.
6. Is an alias different from a label?
7. If you have registered them with the parser (Defining new Units)

Unicode characters for unit symbols

```
public static final String interpunct = String.valueOf(Character.toChars(0x00B7));
public static final String sup0 = String.valueOf(Character.toChars(0x2070));
public static final String sup1 = String.valueOf(Character.toChars(0x00B9));
public static final String sup2 = String.valueOf(Character.toChars(0x00B2));
public static final String sup3 = String.valueOf(Character.toChars(0x00B3));
public static final String sup4 = String.valueOf(Character.toChars(0x2074));
public static final String sup5 = String.valueOf(Character.toChars(0x2075));
public static final String sup6 = String.valueOf(Character.toChars(0x2076));
public static final String sup7 = String.valueOf(Character.toChars(0x2077));
public static final String sup8 = String.valueOf(Character.toChars(0x2078));
public static final String sup9 = String.valueOf(Character.toChars(0x2079));
public static final String supMinus = String.valueOf(Character.toChars(0x207B));
public static final String omega = String.valueOf(Character.toChars(0x03A9));
public static final String micro1 = String.valueOf(Character.toChars(0x00B5));
public static final String micro2 = String.valueOf(Character.toChars(0x03BC));
```

If a particular string will not parse, try creating it using `getQuantity()` then compare the string with the `quantity.toString()` output. You may need to use one of the other parser provided - see [sect-spi], Unit Formatters and Quantity Formatters.

11.2. Formatting

WARNING

According to the Javadocs, SimpleQuantityFormat should be configurable for the number-unit separator and the multiple-radix separators, but I can't get it to work. The default <number><space>[prefix]<unit> works OK, so this is all I will describe until I work out what is going on.

Formatting is an exact reversal of parsing. See the previous table for examples.

In Indriya, the `Quantity.toString()` method delegates to SimpleQuantityFormat. If you want to use one of the other formatters you have to call its format() methods explicitly.

11.2.1. Formatters

SimpleQuantityFormat

The default used for toString() in Indriya. Not locale-sensitive. Number is printed to full precision.

NumberDelimiterQuantityFormat

The decimal number format, separator, and unit format are independently controllable. The best option for presentation to users. Since the number precision is variable, not suitable as a storage format.

Example:

NumberDelimiterQuantityFormat example

```
TBD
```

11.2.2. Locale

TBD

[1] According to the Javadocs, the parser should be configurable for the number-unit separator and the multiple-radix separators, but I can't get it to work. The default <number><space>(prefix)<unit> works OK, so this is all I will describe until I work out what is going on.

Chapter 12. Java Object properties

This section discusses properties of the UOM classes that are not specifically part of the API, but affect how you can use them.

12.1. Extendability

All Unit and Quantity classes are final. They follow the Value-based Class pattern from https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/doc-files/ValueBased.html#Value-basedClasses

12.2. Thread safety

Quantity and Unit instances are immutable therefore unconditionally thread safe.

12.3. Identity, equality and equivalence of Quantitys

To recap, Identity means the same Object (`==`), equality (`equals()`) means strict equality of different objects, and equivalence (`isEquivalentTo()`) means representing the same physical value irrespective of its units.

Identity is obvious, but is rarely used with Objects.

The definition of equality from the Indriya Javadovcs is:

> public boolean equals(Object obj) Compares this quantity against the specified object for strict equality (same unit and same amount). Similarly to the BigDecimal.equals(java.lang.Object) method which consider 2.0 and 2.00 as different objects because of different internal scales, quantities such as Quantities.getQuantity(3.0, KILOGRAM) Quantities.getQuantity(3, KILOGRAM) and Quantities.getQuantity("3 kg") might not be considered equals because of possible differences in their implementations.

The 'might' is a strong indication that the method should not be used, although in fact the example Quantitys are considered equal in Indriya.

Equivalence is the normal test to use: it converts to common units, then compares the values. This lets it work across units from different systems of units, providing they are commensurable.[2]

Caveats:

- The amounts must be truly equal according to (Number): there is no method for "the same {plusminus} a bit".
- The arguments must be typed (eg `Quantity<Mass>` not `Quantity<?>`) which can make the method less convenient to use.

12.4. Ordering

Quantitys are not declared as comparable,[3] but you can use them in Sorted Collections anyway.[4]

Sorting quantities

```
Collection<Quantity<Power>> bag = new TreeSet<>();
bag.add(getQuantity(3.4, KILO_WATT));
bag.add(getQuantity(1.2, WATT));
bag.add(getQuantity(4000, WATT));
bag.add((Quantity) getQuantity(62, WATT)); // Why is this not an error?
// bag.add(getQuantity(3.4, KELVIN)); // Compile time error: incompatible types
System.out.println(bag); // [1.2 W, 62 W, 3.4 kW, 4000 W]
```

Sorting is by base value as you would hope.

If you want to make explicit comparisons, declare or cast your Quantitys to `ComparableQuantity`:

Comparable

```
ComparableQuantity<Power> q1 = getQuantity(3.4, KILO_WATT);
ComparableQuantity<Power> q2 = getQuantity(5.2, WATT);
ComparableQuantity<Power> q3 = getQuantity(4.0, KILO_WATT);
ComparableQuantity<Power> q4 = getQuantity(4000, WATT);
ComparableQuantity<Temperature> q5 = getQuantity(273, KELVIN);

System.out.println(q1.compareTo(q2)); // 1
System.out.println(q3.compareTo(q4)); // 0
System.out.println(q2.compareTo(q4)); // -1
//System.out.println(q2.compareTo(q5)); // Compile time error: Incompatibe types
```

You can use the Comparable.compareTo() directly as shown, but the class also has convenience methods `isLessThan()`, `isGreaterThanOrEqualTo()` etc.

WARNING	The test used in the comparator for "the same" is `isEquivalentTo()` but the `equals()` method is used when adding to Sets. Equals is therefore not consistent with Comparable, and the general contract for Collections will be broken if you use Quantities in Sets. (ref Bloch item X)

12.5. Serialisation

12.5.1. Using java.io.Serializable

`Quantity` is not serializable but `ComparableQuantity` is. Does this mean that you have to hope that QuantityFactories obtained from other libraries produce Serializable as well? Or recreate then using Indriya? Head spinning. The stored size is variable, roughly 20 to 60 bytes.

12.5.2. Using text

Yes, with some caveats.

The text (`Quantity.toString()`) form produced by a formatter is guaranteed to be convertible to and from the binary form without loss of precision. That is:

```
Quantity<Power> q = Quantities.getQuantity(2, KILO_WATT);
Quantity<?> q1 = Quantities.getQuantity(q.toString());
System.out.println(q1.equals(q)); // true, for any q
```

This is required by the JSR (is it?), so should be true for any other implementation as well.

If you have defined your own units, either convert the quantities to built-in units before export, or distribute your program as well, otherwise the recipient will not be able to parse the output.

If you are restricted to ASCII-only files (ie no Unicode), use the ASCII flavour of SimpleQuantityFormat (How? show example)

12.5.3. Using JavaBeans XML

Not currently implemented. See issue #264 and issue #266.

[2] I am at my Java limits here, and making it up

[3] Why not?

[4] Quantities.getQuantity() returns ComparableQuantity which does, and all creation methods seem to delegate to it

Chapter 13. Performance

The advantages of using dimensioned quantities come at a price in speed and storage space. UOM has been reported as about 20 times slower than BigDecimal (issue #298, issue #299).

13.1. Speed

Provisional benchmark results are:

Table 2. Time for arithmetic operations relative to primitive double multiplication

Class	add	subtract	multiply	divide
double	1.0	1.0	1.0 (Ref)	2.0
long	1.2	1.0	1.1	4.3
Double	1.1	1.1	1.1	1.9
BigDecimal	3.0	3.0	3.2	TBD
Quantity	1923.0	1934.0	61.7	2633.0

Table 3. Time for UOM Object operations relative to primitive double multiplication

Operation	Statement	Time
Quantity creation	getQuantity(97.6, WATT)	12.6
Quantity conversion (same units)	quantity.to(KILOWATT)	1.2
Quantity conversion (different units)	quantity.to(KILOWATT)	3254.0
New complex unit	new TransformedUnit<>("°", RADIAN, MultiplyConverter.ofPiExponent(1).concatenate(MultiplyConverter.ofRational(1, 180)))	234.0

Some suggestions for improving speed:

Avoid unnecessary object creation

Define Units and frequently used constants just once:

```
  public static final Unit<Angle> DEGREE_ANGLE = new TransformedUnit<>("\u00b0",
RADIAN, MultiplyConverter.of(180.0 / Math.PI));
  public static final Unit<?> IRRADIANCE = WATT.divide(SQUARE_METRE);
  public static final Unit<Power> KILO_WATT = KILO(WATT);
  public static final Unit<Energy> KILO_WATT_HOUR =
KILO(WATT.multiply(HOUR)).asType(Energy.class);

  public static final Quantity<Time> ZERO_TIME = Quantities.getQuantity(0.0, SECOND);
  public static final Quantity<Energy> ZERO_ENERGY = Quantities.getQuantity(0.0,
JOULE);
  public static final Quantity<Power> ZERO_POWER = Quantities.getQuantity(0.0, WATT);
  public static final Quantity<Dimensionless> ZERO_NUMBER =
Quantities.getQuantity(0.0, ONE);

  public static final Quantity<?> STANDARD_IRRADIANCE = Quantities.getQuantity(1000.0,
IRRADIANCE);
```

Avoid unnecessary Unit conversion

Use consistent units. For long maths routines, this means creating or converting all quantities to consistent units first using `.to()`.

Don't use Quantities (1)

For some tasks, the UnitConverter class can be used to avoid the cost of creating multiple Quantitys (Converting primitives).

Don't use Quantities (2)

In numerically intensive code, use Quantities only at the input and output of your routines, then use primitives to do the actual sums.

```
  Quantity<Power> method(Quantity<Mass> q_input) {
    double q_internal = q_input.to(METRE).getValue().doubleValue();
    ...
    return getQuantity(d_result, WATT);
  }
```

13.2. Size

Table 4. Space requirements comparison

Class	Instance	Bytes	Note
double	Any	8	Fixed
long	Any	8	Fixed
BigDecimal	1923.5	32	Fixed

Class	Instance	Bytes	Note
Quantity	1923.5 W	22	Variable 20-60

Some suggestions for reducing space:

Wrap primitive arrays and collections (memory)

If you have many quantities of the same Unit, you can save space by only storing the Unit once. The following class is type-safe in that attempting to set a quantity that is not commensurable with the unit provided in the constructor will cause a run-time exception (UnconvertibleException).

DimensionedVector.java

```
public class DimensionedVector<T extends Quantity<T>> implements Serializable {

    private final double[] arr;
    private final Unit<T> unit;

    public DimensionedVector(double[] arr, Unit<T> unit) {
        this.arr = arr;
        this.unit = unit;
    }

    // Currently only throws a runtime exception. How can I make it throw a compile
    // time error instead?
    public void set(int i, Quantity<T> qty) {
        arr[i] = qty.to(unit).getValue().doubleValue();
    }

    public Quantity<T> get(int i) {
        return Quantities.getQuantity(arr[i], unit);
    }

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

    @Override
    public boolean equals(Object obj)
    {
        DimensionedVector that = (DimensionedVector)obj;
        return this.unit.equals(that.unit)&&Arrays.equals(this.arr, that.arr);
    }
}
```

Store Quantities as primitives (serialisation)

Each instance of the ThermometerSample class below serialises as 8 bytes, but keeps the type-safety of Quantity.

ThermometerSample.java

```
  public class ThermometerSample implements Serializable {

      /**
       * Stores time in whole seconds, but only visible to clients as a
       * Quantity<Time>. Good for up to 68 years?
       */
      private int time; // 4 bytes
      /**
       * Stores temperature in Kelvin, but only visible to clients as a
       * Quantity<Temperature>
       */
      private float temperature;  // 4 bytes

      public ThermometerSample(Quantity<Time> time, Quantity<Temperature> temperature)
{
          this.time = time.to(Units.SECOND).getValue().intValue();
          this.temperature = temperature.to(Units.KELVIN).getValue().floatValue();
      }

      /**
       * Get the sample time in Seconds
       */
      public ComparableQuantity<Time> getTime() {
          return Quantities.getQuantity(time, Units.SECOND);
      }

      /**
       * Get the temperature in Kelvin
       */
      public ComparableQuantity<Temperature> getTemperature() {
          return Quantities.getQuantity(time, Units.KELVIN);
      }

      @Override
      public boolean equals(Object obj) {
          ThermometerSample that = (ThermometerSample) obj;
          return this.time == that.time && this.temperature == that.temperature;
      }
  }
```

Chapter 14. Using in other JVM languages

Methods allow you to smoothly display code examples in different JVM languages.

14.1. My first method

My first method exposes how to print a message in Java and Kotlin.

{% tabs %} {% tab title="Java" %} Here is how to print a message to `stdout` with Java.

```
System.out.println("My first method");
```

{% endtab %}

{% tab title="Kotlin" %} Here is how to print a message to `stdout` with Kotlin.

```
println("My first method")
```

{% endtab %} {% endtabs %}

Whatever language you are using, the result will be the same.

```
$ My first method
```

Chapter 15. Indriya as part of JSR-365

Indriya is one implementation of JSR-365, and provides a comprehensive set of classes for dealing with SI units. However it does not provide non-SI units or complex formatting. To get these facilities you can use the service provider mechanism contained in the JSR-365 core library.

15.1. Service Provider

The ServiceProvider mechanism is based on the `java.util.ServiceLoader` class (link).

ServiceProvider.available() returns a list of all ServiceProvider instances found in the current classloader(s).

ServiceProvider.current() returns the one with the highest priority.

ServiceProvider.of(name) returns the named ServiceProvider.

The classes that can be obtained through the service provider are:

- UnitFormat
- QuantityFormat
- SystemOFUnits
- Prefix
- QuantityFactory

In each case there is a default class, or you can obtain classes by name.

You can mix and match classes from different service providers, or keep to just one.

Service Provider Interface

```
// Defaults:

FormatService formatService = ServiceProvider.current().getFormatService();

QuantityFormat qFormat = formatService.getQuantityFormat();

UnitFormat uFormat = formatService.getUnitFormat();

...

// Or mix and match by name:

QuantityFormat qFormat =
ServiceProvider.of("otherImplementation").getFormatService().getQuantityFormat("Number
Space");

UnitFormat arabicFormat =
ServiceProvider.of("anotherImplementation").getFormatService().getUnitFormat("arabic")
;

UnitFormat frenchFormat =
ServiceProvider.of("anotherImplementation").getFormatService().getUnitFormat("french")
;

SystemOfUnits imperial =
ServiceProvider.of("otherImplementation").getSystemOfUnitsService().getSystemOfUnits("
imperial");

Set<MetricPrefix> metricPf =
ServiceProvider.of("otherImplementation").getSystemOfUnitsService().getPrefixes(Metric
Prefix.class);

QuantityFactory qfLength =
ServiceProvider.of("otherImplementation").getQuantityFactory(Length.class);

QuantityFactory qfMass =
ServiceProvider.of("otherImplementation").getQuantityFactory(Mass.class);

...
```

Indriya contains several service providers as documented in [appendix-spi].

15.2. Adding to your project

Service providers will be automatically recognised if they are on the class path:

```
        <dependency>
            <groupId>tech.units</groupId>
            <artifactId>indriya</artifactId>
            <version>2.1</version> <!-- 2.1.2 FAILS against si-quantity -->
            <type>jar</type>
            <scope>compile</scope>
        </dependency>
        <!-- https://mvnrepository.com/artifact/si.uom/si-quantity -->

        <dependency>
            <groupId>si.uom</groupId>
            <artifactId>si-quantity</artifactId>
            <version>2.0.1</version>
            <type>jar</type>
            <scope>compile</scope>
        </dependency>

        <dependency>
            <groupId>systems.uom</groupId>
            <artifactId>systems-common</artifactId>
            <version>2.0.2</version>
            <type>jar</type>
            <scope>compile</scope>
        </dependency>

        <dependency>
            <groupId>systems.uom</groupId>
            <artifactId>systems-quantity</artifactId>
            <version>2.0.2</version>
            <type>jar</type>
            <scope>compile</scope>
        </dependency>
```

15.3. Listing services

This code fragment will list the classes available:

```
  public void listSpis() {
      StringBuilder sb = new StringBuilder();
      for (ServiceProvider sp : ServiceProvider.available()) {
          {
              sb.append("\nServiceProvider: " + sp).append("\n");
              for (SystemOfUnits sou :
sp.getSystemOfUnitsService().getAvailableSystemsOfUnits()) {
                  sb.append("  SystemsOfUnits: [" + sou.getName() + "]");
                  for (Unit s : sou.getUnits()) {
                      sb.append(" [" + s.toString() + "]");
                  }
                  sb.append("\n");;
                  sb.append("  Quantity Formatters: ");
                  FormatService formatService = sp.getFormatService();
                  for (String s :
formatService.getAvailableFormatNames(FormatService.FormatType.QUANTITY_FORMAT)) {
                      sb.append(" [").append(s).append("]");
                  }
                  sb.append("\n");
                  sb.append("  Unit Formatters: ");
                  for (String s :
formatService.getAvailableFormatNames(FormatService.FormatType.UNIT_FORMAT)) {
                      sb.append(" [").append(s).append("]");
                  }
                  sb.append("\n");
                  sb.append("  Prefixes: NOT DISCOVERABLE???\n");
              }
          }
      }
      System.out.print(sb.toString());
  }
```

15.4. Using the formatters

Use of the UnitFormat and QuantityFormat classes has already been described. Because they are known only by their Interface, it is not possible to configure them.

15.5. Obtaining units

TBD

15.6. Creating Quantitys

QuantityFactory has a create((Number,Unit) method corresponding to the Quantities.getQuantity(Number,Unit) ???????

Chapter 16. Converting from JScience

16.1. Using quantities

Here's a simple use of a physical quantity with JScience, in this example Length:

```
import org.jscience.physics.amount.Amount;

import javax.measure.quantity.Length;

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

// ...

final Amount<Length> d = Amount.valueOf(214, CENTI(METRE));
final double d_metre = d.doubleValue(METRE);
```

And here's the equivalent code using Units API 2.0 and Indriya:

```
import tech.units.indriya.quantity.Quantities;

import javax.measure.Quantity;
import javax.measure.quantity.Length;

import static javax.measure.MetricPrefix.CENTI;
import static tech.units.indriya.unit.Units.METRE;

// ...

final Quantity<Length> d = Quantities.getQuantity(214, CENTI(METRE));
final double d_metre = d.to(METRE).getValue().doubleValue();
```

16.2. Consistency

While JScience also defines aliases with alternative spellings like METER and constants for many prefixed units like CENTIMETER or MILLIMETER, Indriya encourages consistency and only allows METRE, CENTI(METRE), MILLI(METRE).

16.3. Quantity names

Most quantities have the same names in both projects, but there are some differences:

Amount<Duration> becomes Quantity<Time>

Amount<Velocity> becomes Quantity<Speed>

In these cases Unit API uses the correct SI names, i.e. time and speed. Wikipedia explains the difference between speed and velocity.

16.4. Arithmetic operations

The method names for the elementary arithmetic operations have changed:

plus() becomes add()

minus() becomes subtract()

times() becomes multiply()

Only the method name for division is the same:

divide() is still divide()

However, the runtime exceptions thrown on division by zero are different:

JScience: java.lang.ArithmeticException: / by zero

Indriya: java.lang.IllegalArgumentException: cannot initalize a rational number with divisor equal to ZERO

16.5. Type hints

If you divide or multiply two quantities the Java type system needs a type hint, because it doesn't know the resulting quantity. Here's how this looks in JScience versus Unit API:

With JScience:

```
Amount<Area> a = Amount.valueOf(100, SQUARE_METRE);
Amount<Length> b = Amount.valueOf(10, METRE);
Amount<Length> c = a.divide(b).to(METRE);
```

With Unit API:

```
Quantity<Area> a = Quantities.getQuantity(100, SQUARE_METRE);
Quantity<Length> b = Quantities.getQuantity(10, METRE);
Quantity<Length> c = a.divide(b).asType(Length.class);
```

16.6. Comparing quantities

If you want to compare quantities via compareTo(), isLessThan(), etc. you need quantities of type ComparableQuantity. The Quantities.getQuantity() factory method returns a ComparableQuantity, which is a sub-interface of Quantity.

16.7. Defining custom units

Defining custom units is very similar to JScience. Here's an example for degree (angle), which is not an SI unit:

```
  public static final Unit<Angle> DEGREE_ANGLE =
      new TransformedUnit<>("°", RADIAN,

MultiplyConverter.ofPiExponent(1).concatenate(MultiplyConverter.ofRational(1, 180)));
```

Chapter 17. Error messages

17.1. Compile time errors

This section provides a list of compiler error messages and likely causes.

Compile time error: Incompatible types

CompareTo() with different dimensions, declared Quantity<Power>

17.2. Runtime errors

This section provides a list of exceptions and likely causes.

javax.measure.IncommensurableException: K is not compatible with W

CompareTo with different dimensions, undeclared Quantity

UnconvertibleException

.to()

java.lang.ClassCastException: The unit: W is not compatible with quantities of type interface javax.measure.quantity.Length

Quantity<Length> xordinate = Quantities.getQuantity(3000, WATT).asType(Length.class);

nit<Length> m = AbstractUnit.parse("m").asType(Length.class); Unit<Area> m2 = m.pow(2).asType(Area.class); Unit<Pressure> Pa = NEWTON.divide(m2).asType(Pressure.class); The asType(Class) method, which can be applied on a Unit or Quantity instance, checks at run time if the unit has the dimension of a given quantity, specified as a Class object. If the unit doesn't have the correct dimension, then a ClassCastException is thrown. This check allows for earlier dimension mismatch detection compared to the unchecked casts, which will throw an exception only when a unit conversion is first requested.

IllegalArgumentException

On some arithmetic errors (Arithmetic)

Appendix A: List of Predefined Units

These SI units are available in package `tech.units.indriya.unit.Units`. Units that are required by the JSR have the Unit constant in CAPITALS. Unit constants in lower case are provided in Indriya but may not exist in other implementations.

Quantity type	Dimensions	Unit constant	Symbol	Note
Acceleration	$L \cdot T^{-2}$	METRES_PER_SQUARE_SECOND	m/s2	
AmountOfSubstance	-	MOLE	mol	Base SI unit
Angle	-	RADIAN	rad	
Area	L^2	SQUARE_METRE	m2	
CatalyticActivity	[N]/[T]	KATAL	kat	
Dimensionless	-	ONE	1	
		Percent	%	
ElectricCapacitance	$T^4 \cdot I^2 \cdot L^{-2} \cdot M^{-1}$	FARAD	F	
ElectricCharge	T·I	COULOMB	C	
ElectricConductance	$I^2 \cdot T^3 \cdot L^{-2} \cdot M^{-1}$	SIEMENS	S	
ElectricCurrent	I	AMPERE	A	Base SI unit
ElectricInductance	$L^2 \cdot M \cdot T^{-2} \cdot I^{-2}$	HENRY	H	
ElectricPermittivity	$T^4 \cdot I^2 \cdot L^{-2} \cdot M^{-2}$	FARAD_PER_METRE	F/m	
ElectricPotential	$L^2 \cdot M \cdot T^{-3} \cdot I^{-1}$	VOLT	V	
ElectricResistance	$L^2 \cdot M \cdot T^{-3} \cdot I^{-2}$	OHM	Ω	
Energy	$M \cdot L^2 \cdot T^{-2}$	JOULE	J	
Force	$M \cdot L \cdot T^{-2}$	NEWTON	N	
Frequency	T^{-1}	HERTZ	Hz	
Illuminance	J/L²	LUX	lx	
Length	M	METRE	m	Base SI unit
LuminousFlux	J	LUMEN	lm	
LuminousIntensity	J	CANDELA	cd	Base SI unit
MagneticFieldStrength	$I \cdot L^{-1}$	AMPERE_PER_METRE	A/m	
MagneticFlux	$L^2 \cdot M \cdot T^{-2} \cdot I^{-1}$	WEBER	Wb	

Quantity type	Dimensions	Unit constant	Symbol	Note
MagneticFluxDensity	$M \cdot T^{-2} \cdot I^{-1}$	TESLA	T	
Mass	M	KILOGRAM	kg	Base SI unit
		Gram	g	SI, non-preferred
Power	$L^{2} \cdot M \cdot T^{-3}$	WATT	W	
Pressure	$M \cdot L^{-1} \cdot T^{-2}$	PASCAL	Pa	
RadiationDoseAbsorbed	$L^{2} \cdot T^{-2}$	GRAY	Gy	
RadiationDoseEffective	$L^{2} \cdot T^{-2}$	SIEVERT	Sv	
Radioactivity	T^{-1}	BECQUEREL	Bq	
SolidAngle	-	STERADIAN	sr	
Speed	$L \cdot T^{-1}$	METRES_PER_SECOND	m/s	
Temperature	Θ	KELVIN	K	Base SI unit
		Celsius	°C	SI, non-preferred
Time	T	SECOND	s	Base SI unit
		Minute	min	Non-SI
		Hour	h	Non-SI
		day	d	Non-SI
		Month	mo	Non-SI
		Week	wk	Non-SI
		Year	y	Non-SI
Volume	L^{3}	CUBIC_METRE	m3	
		Litre	l	SI, non-preferred

Appendix B: Service provider: Default

B.1. Unit Formatters

- "EBNF"
- "Local"
- "ASCII"
- "Default"

B.1.1. "EBNF"

Table 5. Test results

Case	Input	Output
Simple	[m]	m
Prefix	[kV]	kV
Compound	[N·m]	N·m
Inverted	[m/s]	m/s
Inverted powers	[m/s²]	m/s²
Repeated units	[kg/m·m·m]	kg·m
Scientific	[m·s□²]	javax.measure.format.MeasurementParseException: tech.units.indriya.format.TokenException:
Leading 1	[1/s]	1/s
Leading 1 (fudge)	[one/s]	javax.measure.format.MeasurementParseException: tech.units.indriya.format.TokenException:
Multiple units	[m·A·W·h]	m·A·W·h
Parentheses	[N(W·h)]	javax.measure.format.MeasurementParseException: tech.units.indriya.format.TokenException: Encountered " "(" "("" at line 1, column 2. Was expecting one of: <EOF> "+" ... "-" ... "*" ... "\u00b7" ... "/" ... "^" ... ":" ... <SUPERSCRIPT_INTEGER> ...
Ohm	[Ω]	Ω
mOhm	[mΩ]	mΩ

Case	Input	Output
Micro (0x00B5)	[µV]	µV
Micro (0x03BC)	[µV]	javax.measure.format.MeasurementParseException: tech.units.indriya.format.TokenException:
Ascii simple	[Nm]	javax.measure.format.MeasurementParseException: tech.units.indriya.format.TokenException:
Ascii simple	[N m]	java.lang.IllegalArgumentException: Lexical error at line 1, column 2. Encountered: " " (32), after : ""
Ascii symbol	[Ohm]	javax.measure.format.MeasurementParseException: tech.units.indriya.format.TokenException:
Ascii prefix	[microOhm]	javax.measure.format.MeasurementParseException: tech.units.indriya.format.TokenException:

B.1.2. "Local"

Table 6. Test results

Case	Input	Output
Simple	[m]	m
Prefix	[kV]	kV
Compound	[N·m]	N·m
Inverted	[m/s]	m/s
Inverted powers	[m/s²]	m/s²
Repeated units	[kg/m·m·m]	kg·m
Scientific	[m·s²]	java.lang.IllegalArgumentException: tech.units.indriya.format.TokenException:
Leading 1	[1/s]	1/s

Case	Input	Output
Leading 1 (fudge)	[one/s]	java.lang.IllegalArgumentException: tech.units.indriya.format.TokenException:
Multiple units	[m·A·W·h]	m·A·W·h
Parentheses	[N(W·h)]	java.lang.IllegalArgumentException: tech.units.indriya.format.TokenException: Encountered " "(" "("" at line 1, column 2. Was expecting one of: <EOF> "+" ... "-" ... "*" ... "\u00b7" ... "/" ... "^" ... ":" ... <SUPERSCRIPT_INTEGER> ...
Ohm	[Ω]	Ω
mOhm	[mΩ]	mΩ
Micro (0x00B5)	[µV]	µV
Micro (0x03BC)	[µV]	java.lang.IllegalArgumentException: tech.units.indriya.format.TokenException:
Ascii simple	[Nm]	java.lang.IllegalArgumentException: tech.units.indriya.format.TokenException:
Ascii simple	[N m]	javax.measure.format.MeasurementParseException: tech.units.indriya.format.TokenMgrError: Lexical error at line 1, column 2. Encountered: " " (32), after : ""
Ascii symbol	[Ohm]	java.lang.IllegalArgumentException: tech.units.indriya.format.TokenException:
Ascii prefix	[microOhm]	java.lang.IllegalArgumentException: tech.units.indriya.format.TokenException:

B.1.3. "ASCII"

Table 7. Test results

Case	Input	Output
Simple	[m]	m
Prefix	[kV]	kV
Compound	[N·m]	N·m
Inverted	[m/s]	m/s
Inverted powers	[m/s²]	m/s²
Repeated units	[kg/m·m·m]	kg/m³
Scientific	[m·s□²]	javax.measure.format.MeasurementParseException: Parse Error
Leading 1	[1/s]	javax.measure.format.MeasurementParseException: unexpected token INTEGER
Leading 1 (fudge)	[one/s]	1/s
Multiple units	[m·A·W·h]	m·A·W·h
Parentheses	[N(W·h)]	javax.measure.format.MeasurementParseException: unexpected token OPEN_PAREN
Ohm	[Ω]	Ω
mOhm	[mΩ]	mΩ
Micro (0x00B5)	[µV]	µV
Micro (0x03BC)	[µV]	µV
Ascii simple	[Nm]	javax.measure.format.MeasurementParseException: Parse Error
Ascii simple	[N m]	javax.measure.format.MeasurementParseException: unexpected token IDENTIFIER
Ascii symbol	[Ohm]	Ω
Ascii prefix	[microOhm]	µΩ

B.1.4. "Default"

Table 8. Test results

Case	Input	Output
Simple	[m]	m

Case	Input	Output
Prefix	[kV]	kV
Compound	[N·m]	N·m
Inverted	[m/s]	m/s
Inverted powers	[m/s²]	m/s²
Repeated units	[kg/m·m·m]	kg/m³
Scientific	[m·s□²]	javax.measure.format.MeasurementParseException: Parse Error
Leading 1	[1/s]	javax.measure.format.MeasurementParseException: unexpected token INTEGER
Leading 1 (fudge)	[one/s]	1/s
Multiple units	[m·A·W·h]	m·A·W·h
Parentheses	[N(W·h)]	javax.measure.format.MeasurementParseException: unexpected token OPEN_PAREN
Ohm	[Ω]	Ω
mOhm	[mΩ]	mΩ
Micro (0x00B5)	[µV]	µV
Micro (0x03BC)	[µV]	µV
Ascii simple	[Nm]	javax.measure.format.MeasurementParseException: Parse Error
Ascii simple	[N m]	javax.measure.format.MeasurementParseException: unexpected token IDENTIFIER
Ascii symbol	[Ohm]	Ω
Ascii prefix	[microOhm]	javax.measure.format.MeasurementParseException: Parse Error

B.2. Quantity Formatters

- "EBNF"
- "Local"
- "NumberDelimiter"
- "Simple"

B.2.1. "EBNF"

Table 9. Test results

Case	Input	Output
Simple	[6 m]	6 m
Extra space	[6 m]	javax.measure.UnconvertibleException: javax.measure.IncommensurableException: one is not compatible with m
Leading space	[` 6 m`]	java.lang.IllegalArgumentException: Number cannot be parsed
Trailing space	[`6 m `]	6 m
Tab	[6 m]	java.lang.IllegalArgumentException: No Unit found
Extra space	[6 m]	javax.measure.UnconvertibleException: javax.measure.IncommensurableException: one is not compatible with m
Prefix	[11.3 kV]	11.3 kV
Rational number	[-5÷3 m]	-5 m
Compound	[11 N·m]	11 N·m
Inverted	[6 m/s]	6 m/s
Inverted powers	[6 m/s²]	6 m/s²
Repeated units	[1013 kg/m·m·m]	javax.measure.UnconvertibleException: javax.measure.IncommensurableException: kg·m is not compatible with kg/m³
Scientific	[6 m·s□²]	javax.measure.format.MeasurementParseException: tech.units.indriya.format.TokenException:
Leading 1	[6 1/s]	6 1/s

B.2.2. "Local"

Table 10. Test results

Case	Input	Output
Simple	[6 m]	6 m

Case	Input	Output
Extra space	[6 m]	javax.measure.UnconvertibleException: javax.measure.IncommensurableException: one is not compatible with m
Leading space	[` 6 m`]	java.lang.IllegalArgumentException: Number cannot be parsed
Trailing space	[`6 m `]	6 m
Tab	[6 m]	java.lang.IllegalArgumentException: No Unit found
Extra space	[6 m]	javax.measure.UnconvertibleException: javax.measure.IncommensurableException: one is not compatible with m
Prefix	[11.3 kV]	11.3 kV
Rational number	[-5÷3 m]	-5 m
Compound	[11 N·m]	11 N·m
Inverted	[6 m/s]	6 m/s
Inverted powers	[6 m/s²]	6 m/s²
Repeated units	[1013 kg/m·m·m]	javax.measure.UnconvertibleException: javax.measure.IncommensurableException: kg·m is not compatible with kg/m³
Scientific	[6 m·s²]	java.lang.IllegalArgumentException: tech.units.indriya.format.TokenException:
Leading 1	[6 1/s]	6 1/s

B.2.3. "NumberDelimiter"

Table 11. Test results

Case	Input	Output
Simple	[6 m]	6 m

Case	Input	Output
Extra space	[6 m]	javax.measure.UnconvertibleException: javax.measure.IncommensurableException: one is not compatible with m
Leading space	[` 6 m`]	java.lang.IllegalArgumentException: Number cannot be parsed
Trailing space	[`6 m `]	6 m
Tab	[6 m]	java.lang.IllegalArgumentException: No Unit found
Extra space	[6 m]	javax.measure.UnconvertibleException: javax.measure.IncommensurableException: one is not compatible with m
Prefix	[11.3 kV]	11.3 kV
Rational number	[-5÷3 m]	-5 m
Compound	[11 N·m]	11 N·m
Inverted	[6 m/s]	6 m/s
Inverted powers	[6 m/s²]	6 m/s²
Repeated units	[1013 kg/m·m·m]	1013 kg/m³
Scientific	[6 m·s□²]	javax.measure.format.MeasurementParseException: Parse Error
Leading 1	[6 1/s]	javax.measure.format.MeasurementParseException: unexpected token INTEGER

B.2.4. "Simple"

Table 12. Test results

Case	Input	Output
Simple	[6 m]	6 m
Extra space	[6 m]	6 m
Leading space	[` 6 m`]	6 m
Trailing space	[`6 m `]	6 m
Tab	[6 m]	6 m
Extra space	[6 m]	6 m

Case	Input	Output
Prefix	[11.3 kV]	11.3 kV
Rational number	[-5÷3 m]	-1.6666666666666666666666666666666666667 m
Compound	[11 N·m]	11 N·m
Inverted	[6 m/s]	6 m/s
Inverted powers	[6 m/s²]	6 m/s²
Repeated units	[1013 kg/m·m·m]	1013 kg/m³
Scientific	[6 m·s²]	javax.measure.format.MeasurementParseException: Parse Error
Leading 1	[6 1/s]	javax.measure.format.MeasurementParseException: unexpected token INTEGER

B.3. System Of Units: Units

Table 13. Units provided by SystemOfUnits Units

Name	Symbol	Dimension	Base units	System unit
Siemens	S	[I]²·[T]³/([L]²·[M])	{A=1, V=-1}	S
null		one	{}	one
Percent	%	one	{}	one
Lux	lx	[J]/[L]²	{lm=1, m=-2}	lx
Litre	l	[L]³	{m=3}	
null	null	[L]²	{m=2}	m²
Sievert	Sv	[L]²/[T]²	{J=1, kg=-1}	Sv
Hour	h	[T]	null	s
Year	y	[T]	null	s
Metre	m	[L]	null	m
Newton	N	[L]·[M]/[T]²	{m=1, kg=1, s=-2}	N
Candela	cd	[J]	null	cd
Kilogram	kg	[M]	null	kg
Second	s	[T]	null	s
Watt	W	[L]²·[M]/[T]³	{J=1, s=-1}	W
Tesla	T	[M]/([T]²·[I])	{Wb=1, m=-2}	T
Katal	kat	[N]/[T]	{mol=1, s=-1}	kat
Volt	V	[L]²·[M]/([T]³·[I])	{W=1, A=-1}	V

Name	Symbol	Dimension	Base units	System unit
Month	mo	[T]	null	s
Kelvin	K	[Θ]	null	K
null	null	[L]/[T]	{m=1, s=-1}	m/s
Ohm	Ω	[L]²·[M]/([T]³·[I]²)	{V=1, A=-1}	Ω
null	null	[L]/[T]	{m=1, s=-1}	m/s
Mole	mol	[N]	null	mol
null	null	[M]	null	kg
Day	d	[T]	null	s
Coulomb	C	[T]·[I]	{s=1, A=1}	C
null	null	[L]³	{m=3}	□
Week	wk	[T]	null	s
Ampere	A	[I]	null	A
Weber	Wb	[L]²·[M]/([T]²·[I])	{V=1, s=1}	Wb
Henry	H	[L]²·[M]/([T]²·[I]²)	{Wb=1, A=-1}	H
Hertz	Hz	1/[T]	{s=-1}	Hz
Pascal	Pa	[M]/([L]·[T]²)	{N=1, m=-2}	Pa
Minute	min	[T]	null	s
Becquerel	Bq	1/[T]	{s=-1}	Bq
Radian	rad	one	{}	rad
Celsius	□	[Θ]	null	K
null	null	[L]/[T]²	{m=1, s=-2}	m/s²
Lumen	lm	[J]	{cd=1, sr=1}	lm
Gray	Gy	[L]²/[T]²	{J=1, kg=-1}	Gy
Joule	J	[L]²·[M]/[T]²	{N=1, m=1}	J
Farad	F	[T]^4·[I]²/([L]²·[M])	{C=1, V=-1}	F
Steradian	sr	one	{}	sr

B.4. Quantity Factories

Length: :tech.units.indriya.quantity.DefaultQuantityFactory <javax.measure.quantity.Length>

Mass: :tech.units.indriya.quantity.DefaultQuantityFactory <javax.measure.quantity.Mass>

etc

B.5. Prefixes: Metric

Name	Symbol	Exponent
YOTTA	Y	24
ZETTA	Z	21
EXA	E	18
PETA	P	15
TERA	T	12
GIGA	G	9
MEGA	M	6
KILO	k	3
HECTO	h	2
DEKA	da	1
DECI	d	-1
CENTI	c	-2
MILLI	m	-3
MICRO	µ	-6
NANO	n	-9
PICO	p	-12
FEMTO	f	-15
ATTO	a	-18
ZEPTO	z	-21
YOCTO	y	-24

B.6. Prefixes: Binary

Name	Symbol	Exponent
KIBI	Ki	1
MEBI	Mi	2
GIBI	Gi	3
TEBI	Ti	4
PEBI	Pi	5
EXBI	Ei	6
ZEBI	Zi	7
YOBI	Yi	8

Appendix C: Service provider: SI

NOTE	Prefixes, QuantityFormat and UnitFormat are the same as Default.

C.1. System Of Units: Non-SI Units

Table 14. Units provided by SystemOfUnits Non-SI Units

Name	Symbol	Dimension	Base units	System unit
null	null	[Θ]	null	K
Unified atomic mass	null	[M]	null	kg
Bohr Radius	null	[L]	null	m
Bar	null	[M]/([L]·[T]²)	{N=1, m=-2}	Pa
null	null	[L]/[T]²	{cm=1, s=-2}	m/s²
null	null	1/[T]	{s=-1}	Bq
null	null	[L]/[T]²	{m=1, s=-2}	m/s²
null	null	one	{}	rad
null	null	1/[T]	{s=-1}	Bq
Electron Volt	null	[L]²·[M]/[T]²	{N=1, m=1}	J
Knot	null	[L]/[T]	{nmi=1, h=-1}	m/s
null	null	[M]	null	kg
null	null	[M]/([L]·[T])	{g=1, cm=-1, s=-1}	kg/(m·s)
null	null	[I]	{daA/4=1, one*3.141592653589793=1}	A
Ångström	null	[L]	null	m
null	null	[T]	null	s
Hectare	null	[L]²	{m=2}	m²
Rad	null	[L]²/[T]²	{J=1, kg=-1}	Gy
null	null	[L]²·[M]/[T]³	{J=1, s=-1}	W
null	null	[M]/([L]·[T]²)	{N=1, m=-2}	Pa
null	null	[L]²·[M]/([T]²·[I])	{V=1, s=1}	Wb
Oersted	null	[I]/[L]	{(A/m)*250=1, one*3.141592653589793=-1}	A/m
null	null	[T]·[I]	{s=1, A=1}	C
Nautical mile	null	[L]	null	m

Name	Symbol	Dimension	Base units	System unit
null	null	[J]/[L]²	{cd=1, cm=-2}	cd/m²
null	null	[N]	null	mol
null	null	[L]²·[M]/[T]²	{N=1, m=1}	J
Roentgen	null	[T]·[I]/[M]	{C=1, kg=-1}	C/kg
null	null	[L]·[M]/[T]²	{m=1, kg=1, s=-2}	N
null	null	[T]	null	s
null	null	[T]	null	s
Astronomical Unit	null	[L]	null	m
Light year	null	[L]	null	m
null	null	[L]/[T]	{m=1, s=-1}	m/s
null	null	1/[T]	{s=-1}	1/s
null	null	[M]/([L]·[T]²)	{N=1, m=-2}	Pa
Second Angle	null	one	{}	rad
null	null	[M]	null	kg
Bel	null	one	{}	one
Dyne	null	[L]·[M]/[T]²	{m=1, kg=1, s=-2}	N
null	null	[L]²/[T]	{cm=2, s=-1}	m²/s
null	null	one	{}	one
null	null	[L]	null	m
Dalton	null	[M]	null	kg
null	null	one	{sr*4=1, one*3.141592653589793=1}	sr
Phot	null	[J]/[L]²	{lm=1, m=-2}	lx
null	null	[M]/([T]²·[I])	{Wb=1, m=-2}	T
Neper	null	one	{}	one
null	null	[T]	null	s
Minute Angle	null	one	{}	rad
null	null	[L]²	{fm=2}	m²
null	null	[T]·[I]	{s=1, A=1}	C
null	null	[L]²/[T]²	{J=1, kg=-1}	Sv
null	null	[J]/[L]²	{cd=1, cm=-2, one*3.141592653589793=-1}	cd/m²

Name	Symbol	Dimension	Base units	System unit
Degree Angle	null	one	{}	rad
Tonne	null	[M]	null	kg

C.2. System Of Units: SI

Table 15. Units provided by SystemOfUnits SI

Name	Symbol	Dimension	Base units	System unit
Radian per second	null	1/[T]	{rad=1, s=-1}	rad/s
Unified atomic mass	null	[M]	null	kg
null	null	[T]·[I]	{s=1, A=1}	C
null	At	[I]	null	At
null	null	[L]²·[M]/[T]³	{W=1, sr=-1}	W/sr
null	null	[L]²·[M]/[T]	{J=1, s=1}	J·s
null	null	[J]/[L]²	{cd=1, m=-2}	cd/m²
null	null	[L]²·[M]/[T]	{J=1, s=1}	J·s
null	null	[M]/[T]³	{W=1, m=-2}	W/m²
null	null	one	{}	rad
null	null	[L]²·[M]/([T]²·[Θ])	{J=1, K=-1}	J/K
null	null	[L]·[M]/([T]²·[I]²)	{N=1, A=-2}	N/A²
null	null	[T]·[I]/[M]	{C=1, kg=-1}	C/kg
null	null	1/[N]	{mol=-1}	m-1
null	ε	[T]^4·[I]²/([L]³·[M])	{F=1, m=-1}	ε
null	null	[L]	null	m
null	null	[M]/([L]·[T])	{Pa=1, s=1}	Pa·s
null	null	[L]²·[M]/[T]²	{N=1, m=1}	J
null	null	1/[L]	{m=-1}	1/m
null	null	[L]/[T]²	{m=1, s=-2}	m/s²
null	null	[L]²/[T]	{m=2, s=-1}	m²/s
null	null	[M]/[T]³	{W=1, sr=-1, m=-2}	W/(sr·m²)
null	null	[I]/[L]	{A=1, m=-1}	A/m
Radian per square second	null	1/[T]²	{rad=1, s=-2}	rad/s²

C.3. Quantity Factories

Length: :tech.units.indriya.quantity.DefaultQuantityFactory <javax.measure.quantity.Length>

Mass: :tech.units.indriya.quantity.DefaultQuantityFactory <javax.measure.quantity.Mass>

etc

Appendix D: Service provider: Common

NOTE	QuantityFormat and UnitFormat are the same as Default.

D.1. System Of Units: United States Customary Units

Table 16. Units provided by SystemOfUnits United States Customary Units

Name	Symbol	Dimension	Base units	System unit
null	null	[Θ]	null	K
null	null	[T]	null	s
null	null	[L]³	{in=3}	□
Cup	null	[L]³	{in=3}	□
null	null	one	{}	rad
Liter	null	[L]³	{m=3}	□
null	null	one	{}	rad
Acre-foot	null	[L]³	{in=3}	□
Metre	m	[L]	null	m
null	null	[L]²	{ft=2}	m²
null	null	one	{}	rad
Liquid Gill	null	[L]³	{in=3}	□
Light year	null	[L]	null	m
null	null	[Θ]	null	K
Tablespoon	null	[L]³	{m=3}	□
Barrel	null	[L]³	{m=3}	□
Hectare	null	[L]²	{m=2}	m²
Minim	null	[L]³	{m=3}	□
Teaspoon	null	[L]³	{m=3}	□
Foot	null	[L]	null	m
Acre	null	[L]²	{ft=2}	m²
Pint	null	[L]³	{in=3}	□
Fluid Ounze	null	[L]³	{in=3}	□
Yard	null	[L]	null	m
Pound	null	[M]	null	kg
US Survey foot	null	[L]	null	m
Mile per hour	null	[L]/[T]	{mi=1, min*60=-1}	m/s

Name	Symbol	Dimension	Base units	System unit
null	null	[L]³	{in=3}	□
Horsepower	null	[L]²·[M]/[T]³	{J=1, s=-1}	W
null	null	[M]	null	kg
null	null	[M]	null	kg
Fluid dram	null	[L]³	{m=3}	□
null	null	one	{}	rad
Are	null	[L]²	{m=2}	m²
Mile	null	[L]	null	m
null	null	one	{}	rad
US gallon	null	[L]³	{in=3}	□
Nautical mile	null	[L]	null	m
null	null	[T]	null	s
null	null	[L]/[T]	{ft=1, s=-1}	m/s
US dry gallon	null	[L]³	{in=3}	□
Knot	null	[L]/[T]	{nmi=1, min*60=-1}	m/s
null	null	one	{}	rad
Inch	null	[L]	null	m
Electron Volt	null	[L]²·[M]/[T]²	{N=1, m=1}	J

D.2. System Of Units: Imperial

Table 17. Units provided by SystemOfUnits Imperial

Name	Symbol	Dimension	Base units	System unit
null	null	[Θ]	null	K
Quart	null	[L]³	{m=3}	□
null	null	[T]	null	s
Acre	null	[L]²	{ft=2}	m²
Cubic Inch	null	[L]³	{in=3}	□
Gill	null	[L]³	{m=3}	□
null	null	[L]³	{m=3}	□
null	null	[L]·[M]/[T]²	{m=1, kg=1, s=-2}	N
Pound	null	[M]	null	kg
null	null	[L]³	{m=3}	□

Name	Symbol	Dimension	Base units	System unit
null	null	[L]²	{ft=2}	m²
null	null	[M]	null	kg
null	null	[M]	null	kg
null	null	[Θ]	null	K
null	null	[L]³	{m=3}	□
null	null	[T]	null	s
null	null	[L]³	{m=3}	□
null	null	[L]³	{m=3}	□
Inch	null	[L]	null	m
null	null	[M]	null	kg
Pint	null	[L]³	{m=3}	□
Minim	null	[L]³	{m=3}	□
null	null	[M]	null	kg
null	null	[L]·[M]/[T]²	{m=1, kg=1, s=-2}	N

D.3. System Of Units: Centimetre–gram–second System of Units

Table 18. Units provided by SystemOfUnits Centimetre–gram–second System of Units

Name	Symbol	Dimension	Base units	System unit
null	null	[M]	null	kg
Kayser	null	1/[L]	{cm=-1}	1/m
Gal	null	[L]/[T]²	{cm=1, s=-2}	m/s²
Dyne	null	[L]·[M]/[T]²	{m=1, kg=1, s=-2}	N
Stokes	null	[L]²/[T]	{cm=2, s=-1}	m²/s
null	null	[L]	null	m
Erg	null	[L]²·[M]/[T]²	{N=1, m=1}	J
Erg per second	null	[L]²·[M]/[T]³	{erg=1, s=-1}	J/s
centimetre per second	null	[L]/[T]	{cm=1, s=-1}	m/s
Barye	null	[M]/([L]·[T]²)	{N=1, m=-2}	Pa
Poise	null	[M]/([L]·[T])	{g=1, cm=-1, s=-1}	kg/(m·s)
Second	s	[T]	null	s

D.4. Quantity Factories

Length: :tech.units.indriya.quantity.DefaultQuantityFactory <javax.measure.quantity.Length>

Mass: :tech.units.indriya.quantity.DefaultQuantityFactory <javax.measure.quantity.Mass>

etc

D.5. Prefixes: Indian

Name	Symbol	Exponent
EK	E	1
DAS	D	1
SAU	S	2
SAHASR	SA	3
LAKH	Lk	5
CRORE	cr	7
ARAWB	A	9
KHARAWB	K	11
NEEL	N	13
PADMA	Pa	15
SHANKH	SH	17
MAHASHANKH	M	19

D.6. Prefixes: Tamil

Name	Symbol	Exponent
PATU	P	1
nūru	S	2
āyiram	SA	3
pattāyiram	Lk	4
nūraiyiram	Cr	5
meiyyiram	A	6
tollun	K	9
īkiyam	N	12
neḷai	Pa	15
iḷañci	SH	18
veḷḷam	M	20

Name	Symbol	Exponent
āmpal	M	21

D.7. Prefixes: IndianAncient

Name	Symbol	Exponent
EK	E	1
DAS	D	1
SAU	S	2
SAHASR	SA	3
LAKH	Lk	5
CRORE	cr	7
ARAWB	A	9
KHARAWB	K	11
NEEL	N	13
PADMA	Pa	15
SHANKH	SH	17
MAHASHANKH	M	19

D.8. Prefixes: TamilAncient

Name	Symbol	Exponent
PATU	P	1
nūru	S	2
āyiram	SA	3
pattāyiram	Lk	4
nūraiyiram	Cr	5
meiyyiram	A	6
tollun	K	9
īkiyam	N	12
neḷai	Pa	15
iḷañci	SH	18
veḷḷam	M	20
āmpal	M	21

D.9. Prefixes: Verdic

Name	Symbol	Exponent
PATU	P	1
nūru	S	2
āyiram	SA	3
pattāyiram	Lk	4
nūraiyiram	Cr	5
meiyyiram	A	6
tollun	K	9
īkiyam	N	12
neḷai	Pa	15
iḷañci	SH	18
veḻḻam	M	20
āmpal	M	21