Solutions to Quiz 2 (April 27, 2018)

This quiz uses the same abstract data type as Quiz 1.

In sciences like physics and chemistry, *dimensional analysis* is often used to check calculations for errors.

Dimensional analysis associates a number with a unit of measure, called its *dimension*. Examples of dimensioned numbers include:

- a length of 10 meters (m)
- a time of 0.8 seconds (s)
- a relative velocity of -20 meters per second (m/s, or ms^{-1})
- an acceleration of 9.8 meters per second per second (m/s^2)
- an area of 100 square meters (m^2)
- a volume change of -0.001 cubic meters (m^3)

Less familiar but still valid examples of dimensioned numbers include:

- 99.9 seconds per meter (s/m)
- 2 meter-seconds ($m \cdot s$)

A number's dimension can be empty, or *dimensionless*, like π.

Two rules of dimensional analysis are:

- Numbers with different dimensions should not be added, subtracted, or compared. For example, it makes no sense to compare a length in meters with a time in seconds, or to add them together.
- The ratio (or product) of two dimensioned numbers produces a number whose dimension is the ratio (or product) of the two dimensions. For example, a velocity in m/s multiplied by a time in seconds produces a length in meters.

The problems in this quiz refer to the code for `DimDouble` on page 5, which you may detach.

Lines 4–11 implement a static creator method, and lines 12–16 declare instance methods of `DimDouble`.

Problem 1 (Grammars) (**22 points**).
Ben Bitdiddle is building a system to manipulate dimensioned numbers. To start, Ben wants to parse a product of dimensioned numbers and produce the correct output.

For example, the input: `4 x 3 kg x 9.8 m / s^2 x 1.5 m x 0.1 / s`

equals 17.64 $kg\ m^2/s^3$, or 17.64 watts.

In the input, *terms* of the product are separated by `'x'` characters. Within a term, if there are units in the denominator, those units all follow a single `'/'`.

The terms in the example above are all *valid*. The following term is not as useful, but is also valid:

`0.0 kg^22 m m / kg m kg^2 s`

Examples of *invalid* terms that the grammar should reject include:

`1 / s^2^2` (no extra unit exponents)
`1.5 m^1` (no unit exponents of 1)

`4 kg^0`	(no unit exponents of 0)
`9.8 m s^-2`	(no negative unit exponents)
`3 kg /`	(no empty unit denominators)
`1 / 10 s`	(no numbers in the denominator)

Complete the grammar by filling in the rest of each incomplete production. **`product`** is the root. The grammar should use all the productions. Make sure we can read what you write. No credit will be given for unreadable or ambiguous writing.

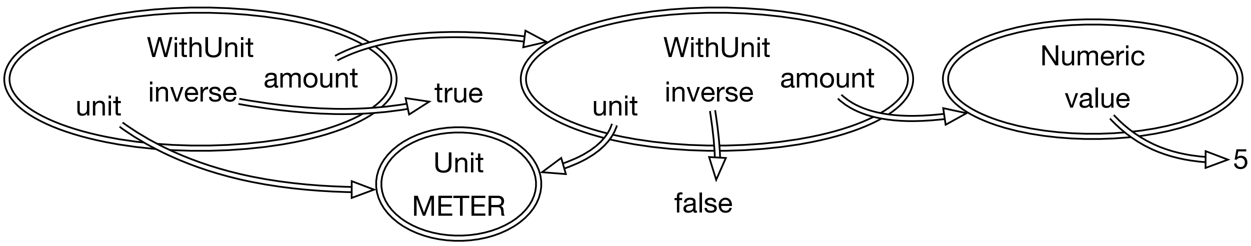
```
@skip whitespace {

  product ::= term ('x' term)* ;

  term  ::=

  numer_units ::= dimension+ ;

  denom_units ::=

}

dimension ::=

exponent ::=

value ::= '-'? \d+ ('.' \d*)? ;
unit ::= 'm' | 'kg' | 's' | 'A' | 'K' | 'mol' | 'cd' ;
whitespace ::= [ \t\r\n]+ ;
```

Solution.

```
term ::= value numer_units? denom_units? ;
denom_units ::= '/' dimension+ ;
dimension ::= unit ('^' exponent)? ;
exponent ::= [2-9] | [1-9] [0-9]+ ;
```

■

Problem 2 (Recursive Types) (**18 points**).
Alyssa proposes the `DimDouble` implementation on page 5, using two concrete variants. She draws a snapshot diagram to show Ben an example:



(a) Write a clear, complete *abstraction function* for **`Numeric`**:

Solution. AF(value) = dimensionless number *value* ■

(b) Write a clear, complete *safety from rep exposure argument* for **`WithUnit`**:

Solution. amount, unit, and inverse are all private final fields and immutable values ■

Implement **`numericValue()`**:

(c) In **`Numeric`**:

```
@Override public double numericValue() {


}
```

Solution. `return value;` ■

(d) In **`WithUnit`**:

```
@Override public double numericValue() {


}
```

Solution. `return this.amount.numericValue();` ■

Problem 3 (Specifications) (**18 points**).
Ben is implementing `DimDouble.`**`units()`** and **`dimensionless()`**. Their signatures, and the spec for `dimensionless()`, are on page 5 (lines 13–15).

Alyssa points to the snapshot diagram in Problem 2. "Ben, make sure this is a valid rep for the dimensionless number 5," she says.

"I'm going to use the same implementation of **`dimensionless()`** in both variants," says Ben. "It doesn't depend on the rep at all because it just calls `units()`:"

```
    @Override public boolean dimensionless() {
        return this.units().isEmpty();
    }
```

"Good plan," says Alyssa, "but it depends on the spec of **`units()`**."

Write two specifications for `units()`, one weaker and one stronger. You may *not* state preconditions, only postconditions.

Ben's implementation should only work with *one* of your specs: the other spec should *not* be enough to guarantee that his implementation is correct.

Then, write an implementation for `dimensionless()` that works with *both* of your specs for `units()`. Similar to Ben's implementation, your code should work for *both* concrete variants of `DimDouble`, by calling `units()` instead of examining the rep.

(a) Weaker spec for **`units()`**:

Solution. returns a map `m` where for all units `u`, `m.getOrDefault(u, 0)` is the exponent of `u` in `this`; 0 indicates unit not present ■

(b) Stronger spec for **`units()`**:

Solution. returns a map `m` where for all units `u` in the units of `this`, `m.get(u)` is the exponent of `u`; no value in the map is 0 ■

(c) **`dimensionless()`** implementation that works with either `units()` spec:

```
@Override public boolean dimensionless() {


}
```

Solution. `return units().values().stream().allMatch(e -> e == 0);` ■

Problem 4 (Code Review) (**28 points**).
The code for **`parallelMultiply`** on page 6 is buggy. For each of these code review comments, circle whether you AGREE or DISAGREE with the comment, and explain why in one sentence.

(a) Because of `subList`, this code creates aliases to the `terms` input from multiple threads. That means there is a thread safety bug if `terms` is not threadsafe.

Solution. AGREE, line 7 will run on multiple threads, accessing the original `terms` through `subList` wrappers. If `terms` is not threadsafe, the concurrent calls to observers `size`, `get` are not threadsafe. ■

(b) Need to wait for both threads to finish (for example, call `join()` on them) before line 18 where `results` is used.

Solution. DISAGREE, each `take()` will block until another result is available. ■

(c) There is a race condition bug with how this code uses the `results` queue twice on line 18.

Solution. DISAGREE, each `take()` will return a different result, and since multiplication is commutative, they can be multiplied in either order. ■

(d) Need to return a defensive copy on line 7 to prevent an aliasing bug.

Solution. DISAGREE, `DimDouble` is immutable, so aliasing is not a problem. ■

Problem 5 (Concurrency) (**14 points**).
Assume we fix any issues from the previous question, and we run:

```
public static void main(String[] args) {
    DimDouble someMeters = DimDouble.make(9.8, new Unit[] { METER }, new Unit[] {});
    DimDouble perSecond = DimDouble.make(1, new Unit[] {}, new Unit[] { SECOND });
    // 9.8 m  x  1 / s  x  1 / s  =  9.8 m / s^2
    DimDouble g = parallelMultiply(Arrays.asList(someMeters, perSecond, perSecond));
}
```

(a) How many **`main`** threads will be created?

Solution. 1 ■

(b) And how many additional threads will be **created by `parallelMultiply`**?

Solution. 4 ■

(c) In total, how many threads will multiply a pair of `DimDouble`s?

Solution. 2 ■

(d) Leif Noad realizes that we can implement `parallelMultiply` using Java's parallel streams.

Complete the implementation using the two-argument `reduce` method whose signature is:

$$\text{reduce} : \text{Stream}<\text{T}> \times \text{T} \times (\text{T} \times \text{T} \to \text{T}) \to \text{T}$$

```
public static DimDouble parallelMultiply(List<DimDouble> terms) {
    return terms.stream().parallel().reduce(

        ,

        );

}
```

Solution. `DimDouble.ONE, DimDouble::multiply` ■

You may detach this page. Write your username at the top, and hand in all pages when you leave.

```
public enum Unit { METER, KILOGRAM, SECOND, AMPERE, KELVIN, MOLE, CANDELA }
```

```
/** Immutable dimensioned number. */
public interface DimDouble {
    public static final DimDouble ONE = make(1, new Unit[] {}, new Unit[] {});

    public static DimDouble make(double value, Unit[] numeratorUnits,
                                               Unit[] denominatorUnits) {
        DimDouble amount = new Numeric(value);
        for (Unit unit : numeratorUnits)
            amount = new WithUnit(amount, unit, false);
        for (Unit unit : denominatorUnits)
            amount = new WithUnit(amount, unit, true);
        return amount;
    }

    public double numericValue();

    public Map<Unit,Integer> units();
```

```java
    /** @return true if and only if this DimDouble is a dimensionless value */
    public boolean dimensionless();

    public DimDouble multiply(DimDouble other);
}

public class Numeric implements DimDouble {
    private final double value;

    public Numeric(double value) { this.value = value; }

    // ... methods ...
}

public class WithUnit implements DimDouble {
    private final DimDouble amount;
    private final Unit unit;
    private final boolean inverse;

    public WithUnit(DimDouble amount, Unit unit, boolean inverse) { ... }

    // ... methods ...
}
```

```java
public class Parallel {

    public static DimDouble parallelMultiply(List<DimDouble> terms) {
        final int termCount = terms.size();

        // product of no terms is the identity
        if (termCount == 0) { return DimDouble.ONE; }
        // product of one term is itself
        if (termCount == 1) { return terms.get(0); }

        // split 'terms' in half
        final List<DimDouble> firstHalf = terms.subList(0, termCount/2);
        final List<DimDouble> secondHalf = terms.subList(termCount/2, termCount);
        final List<Thread> threads = new ArrayList<>();
        final BlockingQueue<DimDouble> results = new LinkedBlockingQueue<>();

        // recursively compute products for each half in parallel...
        threads.add(new Thread(() -> results.put(parallelMultiply(firstHalf))));
        threads.add(new Thread(() -> results.put(parallelMultiply(secondHalf))));
        for (Thread t : threads) { t.start(); }

        // ... and multiply those two products
        return results.take().multiply(results.take());
    }
}
```

For reference:

`List.subList` (used on lines 9 & 10) creates a wrapper that refers to part of a larger list. Its spec is:

> `List<E> subList(int fromIndex, int toIndex)`
>
> Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive. (If `fromIndex` and `toIndex` are equal, the returned list is empty.) The returned list is backed by this list.
>
> For example, the following removes a range of elements from a list:
> `list.subList(from, to).clear();`

`BlockingQueue` (used on line 12) provides both non-blocking operations (*e.g.* `add` and `remove`) and blocking operations (*e.g.* `put` and `take`).