Quiz 2 (April 27, 2018)

Your name:____

Your Kerberos username:__________________________________

You have 50 minutes to complete this quiz. It contains 10 pages (including this page) for a total of 100 points.

The quiz is closed-book and closed-notes, but you are allowed one two-sided page of notes.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Before you begin, write your Kerberos username on the top of every page.

Please write neatly. **No credit will be given if we cannot read what you write.**

For questions which require you to choose your answer(s) from a list, do so clearly and unambiguously by circling the letter(s) or entire answer(s). Do not use check marks, underlines, or other annotations – they will not be graded.

Good luck!

Problem	Points
1: Grammars	22
2: Recursive Types	18
3: Specifications	18
4: Code Review	28
5: Concurrency	14
Total	100

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 9, 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`

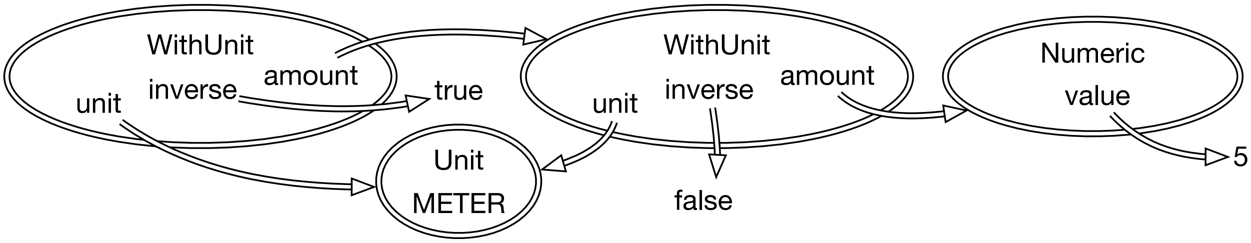
Examples of *invalid* terms that the grammar should reject include:

`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  ::= value

  numer_units ::= dimension+ ;

  denom_units ::= '/'

}

dimension ::= unit

exponent ::= [2-9]

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

Problem 2 (Recursive Types) (**18 points**).
Alyssa proposes the `DimDouble` implementation on page 9, using two concrete variants. She draws a snapshot diagram to show Ben an example:



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

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

Implement **`numericValue()`**:

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

```
@Override public double numericValue() {

}
```

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

```
@Override public double numericValue() {

}
```

Problem 3 (Specifications) (**18 points**).
Ben is implementing `DimDouble.`**`units()`** and **`dimensionless()`**. Their signatures, and the spec for `dimensionless()`, are on page 9 (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()`**:

returns

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

returns

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

```
@Override public boolean dimensionless() {

}
```

Problem 4 (Code Review) (**28 points**).

The code for **`parallelMultiply`** on page 10 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.

AGREE / DISAGREE because:

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

AGREE / DISAGREE because:

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

AGREE / DISAGREE because:

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

AGREE / DISAGREE because:

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? [ ]

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

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

(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:

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

```
public static DimDouble parallelMultiply(List<DimDouble> terms) {
    return terms.stream().parallel().reduce(
            [                                        ],
            [                                        ]);
}
```

```
SFB
ETU
RFC
```

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

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

```java
/** 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();

    /** @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
1 public class Parallel {

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

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

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

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

17        // ... and multiply those two products
18        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`).