Solutions to Quiz 1 (March 23, 2018)

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 6, which you may detach.

The back side of that page has abbreviated specs for Java's `List` and `Map` that may be useful if you need them.

The last problem of this quiz takes longer than the earlier problems. Use your time well.

Problem 1 (Code review) (**20 points**).
The code for **`DimDouble`** 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) Line 12: remove `final` from `result`, because it prevents changing `result` in lines 13–14.

Solution. DISAGREE, `final` prevents reassigning the `result` variable but doesn't prevent mutating the objects it points to. ■

(b) Lines 13–14: this code creates aliasing between the reps of different `DimDouble` objects.

Solution. AGREE, two `DimDouble` objects can now point to the same `ArrayList` objects for their units.

It's not necessarily a bug. Aliasing within the implementation of a type is less harmful than aliasing with a client (which would be rep exposure). ■

(c) Line 28: this code violates the spec of `append`.

Solution. AGREE, the precondition does not exclude the case `list1==list2`, but the assertion may throw an error in that case. ■

(d) Lines 29–30: this code violates the spec of `append`.

Solution. AGREE, this code mutates `list1`, which was not allowed by the spec. ■

Problem 2 (Specs) (**22 points**).
Consider this spec for a method that uses `DimDouble`:

```
/**
 * Compute the radius of a hypersphere, the set of points equidistant
 * from a center point in N-dimensional space.  For example, a 3-sphere is
 * a conventional sphere in 3D, and a 2-sphere is a circle in 2D.
 *
 * @param volume must be > 0 with units in meters^N for N > 0
 * @return the radius of an N-sphere with the given volume, in meters
 */
public DimDouble radiusOfHyperSphere(DimDouble volume)
```

Here is an example of a correct call to this method:

```
radiusOfHyperSphere(new DimDouble(2, METERS))
```

For each of the following *buggy* calls to the method, circle whether the **spec guarantees that** the bug is caught by a static error, by a dynamic error, or no guarantee that the bug is caught.

(a) `radiusOfHyperSphere(21.3, METERS)`

Solution. STATIC ERROR, because `radiusOfHyperSphere` requires one parameter, not two. ■

(b) `radiusOfHyperSphere(new DimDouble(-5, METERS))`

Solution. NO GUARANTEE, because this violates the precondition, so the implementation can do anything. ■

(c) `radiusOfHyperSphere(new DimDouble(7, SECONDS))`

Solution. NO GUARANTEE, because this violates the precondition, so the implementation can do anything. ■

(d) `Math.PI * radiusOfHyperSphere(new DimDouble(0.5, METERS))`

Solution. STATIC ERROR, because `Math.PI` is a `double` but `radiusOfHyperSphere` returns a `DimDouble`. ■

Louis Reasoner proposes several partitions for choosing test cases for `radiusOfHyperSphere()` based on the `volume` parameter and the radius returned. Alyssa likes all of his partitions *except* this one:

```
volume is dimensionless, in meters, or in seconds
```

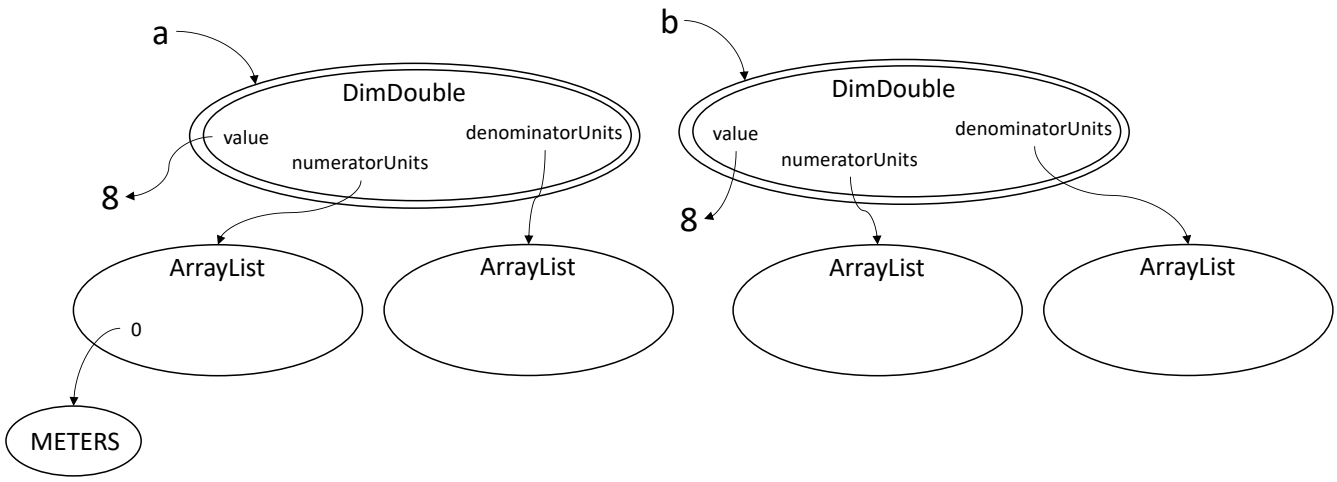
Give two (substantially different) reasons why this proposal is not a good partition. Each reason should be one sentence in its own box below.

(e) Solution. Two parts of the partition (dimensionless and seconds) violate the precondition, so tests can't be written for them. Those parts should be removed. ■

(f) Solution. The partition is not exhaustive – it does not cover the whole legal input space. It needs at least an additional part like "in m^N for $N > 1$". ■

Problem 3 (Equality) (**22 points**)**.**
Ben Bitdiddle points out that `DimDouble` is missing not only specs, but also an abstraction function and rep invariant. The implementations for `multiply()` and `divide()` give some clues about what reps are intended to be legal and how the rep is intended to be interpreted.

(a) Complete the snapshot diagram below to form two different-looking rep values `a` and `b` that should have the same abstract value.



Solution. One possible answer:

- `a.numeratorUnits=[METERS,SECONDS]`
- `a.denominatorUnits=[]`
- `b.numeratorUnits=[SECONDS,METERS]`
- `b.denominatorUnits=[]`

Other possible answer:

- `a.numeratorUnits=[METERS]`
- `a.denominatorUnits=[]`
- `b.numeratorUnits=[METERS,SECONDS]`
- `b.denominatorUnits=[SECONDS]`

Many other solutions are possible. ■

(b) Write the abstract value that your `a` and `b` represent.

Solution. For the two answers to (a) proposed above, the abstract values would be 8 meters·seconds and 8 meters, respectively. ■

(c) Suppose `DimDouble` uses the following helper function as part of implementing its equality operation.

```
private boolean sameAs(DimDouble that) {
    return this.value == that.value
        && this.numeratorUnits  .containsAll(that.numeratorUnits  )
        && this.denominatorUnits.containsAll(that.denominatorUnits);
}
```

Which properties of an equivalence relation does this function satisfy, and which does it violate? Fill in each blank with a name, and circle YES if it's satisfied and NO if not.

_____ YES NO

_____ YES NO

_____ YES NO

Solution. YES to reflexive, because `==` is reflexive, and so is `containsAll`: `list.containsAll(list)` is always true.

NO to symmetric, because `containsAll` is not symmetric. For example, if `a.numeratorUnits` is a strict subset of `b.numeratorUnits` and the other fields are identical, then `a.sameAs(b)` is true but `b.sameAs(a)` is false.

YES to transitive, because `==` is transitive and so is `containsAll`: if `a`'s lists `containsAll` `b`'s corresponding lists, and `b`'s lists `containsAll` `c`'s lists, then `a`'s lists `containsAll` `c`'s lists.

■

Problem 4 (Reps) (**36 points**).
Alyssa Hacker suggests an alternative rep for `DimDouble`:

```
private double value;
private Map<Unit, Integer> units = new HashMap<>();
```

Write an abstraction function, rep invariant, and two operations using this new rep. Read this whole page before making design decisions. To simplify, you can assume for this entire page that the only base units are `METERS` and `SECONDS`.

(a) Abstraction function

Solution.

`AF(value,units)` =

the dimensioned number with value `value` and dimension $\prod_{u \in \texttt{units.keySet()}} u^{\texttt{units.get(}u\texttt{)}}$

Or, if simplified to assume just METERS and SECONDS:

`AF(value,units)` =

the dimensioned number `value` $m^{\texttt{units.get(METERS)}} s^{\texttt{units.get(SECONDS)}}$

Note the abstraction function must map to all possible abstract values described by the "dimensioned number" spec on page 1 including dimensionless numbers, zero-valued numbers, and negative-valued numbers. The new rep must still be able to support all 'DimDouble' operations on page 6. We are just changing the rep, not the spec of the abstract data type.

■

(b) Rep invariant (implicit 6.031 rep invariant can be omitted)

Solution. Many answers are possible, for example:

- true (i.e., no rep invariant beyond the implicit non-null invariant)
- `units.get(u)` $\neq 0$ for all `u` $\in$ `units.keySet()`
- `units.keySet()` is the set of all possible `Unit` values
- `units.keySet()` $=$ `{METERS, SECONDS}` (exploiting the allowed simplification)

Note the rep invariant must allow the abstraction function to map to all abstract values described by the spec. The new rep must still be able to support all 'DimDouble' operations on page 6. Examples of rep invariants that are *wrong*:

- `value` $>= 0$ (wrong because negative-valued numbers need to be represented by the type)
- `units.get(u)` $\in \{1, -1\}$ for all `u` $\in$ `units.keySet()` (wrong because it can't represent meters per second per second)
- `units.get(u)` > 0 for all `u` $\in$ `units.keySet()` (wrong because it can't represent meters per second)

There are also statements that are *unnecessary* to put in a rep invariant comment. Examples include:

- `units` and none of its keys or values are `null` (unnecessary because it's the implicit 6.031 invariant)
- the keys of `units` are `Unit` objects and the values are `Integer` objects (unnecessary because it's stated by the static types of the rep fields, so Java enforces it)
- `units` contains no repeating keys (unnecessary because it's guaranteed by the abstract type `Map`)
- every integer value in `units` is positive, negative, or zero (unnecessary because that's vacuously true for all integers)

■

Complete the two operations below using this new rep. Note that part of the code has been filled in already.

(c)
```
/** make a dimensioned number equal to ‘value‘ with units ‘base‘ */
public DimDouble(double value, Unit base) {
    this.value = value;
```

Solution.

```
    // if rep invariant states that units.keySet() has all possible Units:
    this.units.put(METERS, 0);
    this.units.put(SECONDS, 0);

    // all rep invariants need this line:
    this.units.put(base, 1);
```

■

```
    checkRep();
}
```

(d)
```
/** @return the product of ‘this‘ and ‘that‘ */
public DimDouble multiply(DimDouble that) {
    DimDouble result = new DimDouble(this.value * that.value);
```

Solution.

```
    // if rep invariant states that units.keySet() has all possible Units:
    result.units.put(METERS, this.units.get(METERS) + that.units.get(METERS));
    result.units.put(SECONDS, this.units.get(SECONDS) + that.units.get(SECONDS));

    // otherwise can't assume that anything is mapped:
    result.units.put(METERS,   this.units.getOrDefault(METERS,  0)
                             + that.units.getOrDefault(METERS,  0));
    result.units.put(SECONDS,  this.units.getOrDefault(SECONDS, 0)
                             + that.units.getOrDefault(SECONDS, 0));

    // if rep invariant states that units can only have nonzero values:
    if (result.units.get(METERS) == 0) result.units.remove(METERS);
    if (result.units.get(SECONDS) == 0) result.units.remove(SECONDS);
```

■

```
    result.checkRep();
    return result;
}
```

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

```
public enum Unit { METERS, SECONDS }
```

```
1  /** Immutable dimensioned number. */
2  public class DimDouble {
3      private double value;
4      private List<Unit> numeratorUnits = new ArrayList<>();
5      private List<Unit> denominatorUnits = new ArrayList<>();

6      public DimDouble(double value) {
7          this.value = value;
       }

8      public DimDouble(double value, Unit base) {
9          this.value = value;
10         this.numeratorUnits.add(base);
       }

11     public DimDouble add(DimDouble that) {
12         final DimDouble result = new DimDouble(this.value + that.value);
13         result.numeratorUnits = this.numeratorUnits;
14         result.denominatorUnits = this.denominatorUnits;
15         return result;
       }

16     public DimDouble multiply(DimDouble that) {
17         final DimDouble result = new DimDouble(this.value * that.value);
18         result.numeratorUnits   = append(this.numeratorUnits,   that.numeratorUnits);
19         result.denominatorUnits = append(this.denominatorUnits, that.denominatorUnits);
20         return result;
       }

21     public DimDouble divide(DimDouble that) {
22         final DimDouble result = new DimDouble(this.value / that.value);
23         result.numeratorUnits   = append(this.numeratorUnits,   that.denominatorUnits);
24         result.denominatorUnits = append(this.denominatorUnits, that.numeratorUnits);
25         return result;
       }

26     /** @return a list of the elements of list1 followed by the elements of list2 */
27     private static List<Unit> append(List<Unit> list1, List<Unit> list2) {
28         assert list1 != list2;
29         List<Unit> result = list1;
30         result.addAll(list2);
31         return result;
       }

32     // ... more operations
   }

   interface List<E> {
     /** Modifies this list by appending e to the end of it. */
     public void add(E e);
```

```
  /** Modifies this list by appending the elements of l to the end, in order. */
  public void addAll(List<E> l);

  /** @return the number of elements in this list. */
  public int size();

  /** @return true iff e is an element of this list. */
  public boolean contains(E e);

  /** @return true iff every element of l is an element of this list. */
  public boolean containsAll(List<E> l);

  /** @return the element at position index in this list
      @throws IndexOutOfBoundsException if index is out of range for this list */
  public E get(int index);

  /** Modifies this list by removing the first occurrence of e from this list, if any. */
  public void remove(E e);

  /** Modifies this list by removing all elements in this list that are also in l.*/
  public void removeAll(List<E> l);

  // ... more operations
}

interface Map<K,V> {
  /** Modifies this map by mapping key to value, replacing any previous mapping for key. */
  public void put(K key, V value);

  /** @return the value to which key is mapped, or null if key not mapped. */
  public V get(K key);

  /** @return value to which key is mapped, or defaultValue if key not mapped. */
  public V getOrDefault(K key, V defaultValue);

  /** Modifies this map by removing the value associated with key, if any. */
  public void remove(K key);

  /** @return the number of key-value mappings in this map. */
  public int size();

  /** @return a Set view of the keys contained in this map. */
  public Set<K> keySet();

  // ... more operations
}
```