Homework # 1
due Monday, September 10, 10:00 PM

In this assignment, you will implement simple immutable ADTs for three conceptions of time:

**Duration** Time as a physical quantity. An amount of time. Example: 3 hours.

**Time** Time as a point along a timeline. Example: The signing of the Declaration of Independence (July 4, 1776).

**Period** Time as an extent of time along the timeline. Example: World War II (September 1, 1939 to September 2, 1945)

We provide JUnit test cases to test your code. Some of the tests are “locked” as described in the lab exercise.

1 Immutable ADTs and standard methods

This week, our ADTs will be *immutable*—all fields are final. The fields are assigned in the constructor and never changed. If you find yourself adding non-final fields or wanting to change the value of a field, you are doing something wrong.

Immutable ADTs simplify programming because then you don’t need to worry that (say) an hour suddenly becomes a minute. This is one of many examples where reducing power makes programming clearer and simpler.

In most situations (including this Homework), immutable ADTs should override the standard implementations of the following methods:

- `equals(Object)` Return true if the argument has the same value as this.
- `hashCode()` Return an integer that summarizes this value. This method should be efficient. Do not create a string in the process.
- `toString()` Return a descriptive string.

Additionally, if the ADT represents a linear quantity, it should declare that it implements `Comparable<MySelf>` (with its own class name as a parameter) and then implement the following method:

- `compareTo(MySelf)` Return a negative number if this comes before the argument, zero if they are equal, or a positive number if this comes after the argument. Here `MySelf` should be replaced with the name of the class.

For this homework assignment, `Duration` and `Time` should be comparable, but `Period` should not.

2 Concerning the Duration ADT

A duration is a non-negative amount of time; a duration can be as short as a millisecond, or as long as 100 million years. The zero duration is also possible.

The `Duration` ADT implements the standard methods described above, where `toString()` returns a string given in terms of milliseconds (ms.), seconds (s.), minutes (min.), hours (hr.), days
or years. Units larger than milliseconds are chosen only if the duration is at least as big as the unit. The number in front of the unit is written as a decimal.

The Duration ADT provides the following additional operations:

- **add(Duration)** Return a new duration that is equal in size to the sum of the sizes of this duration and the argument.

- **subtract(Duration)** Return a new duration that when added to the argument will be equal to this. If no such duration exists (because the argument is bigger than this) throw an “arithmetic exception.”

- **scale(double)** Return a new duration that is the argument times bigger than this duration. For example, scaling an hour by 24 gets a day; scaling an hour by 0.5 gets the same duration as scaling a minute by 30.

- **divide(Duration)** Return the relative size of one duration to another as a double (number). For example, DAY.divide(HOUR) gives 24 and SECOND.divide(MILLISECOND) gives 1000.

In lieu of a public constructor, the Duration class provides the following constants:

- INSTANTANEOUS (zero-length duration), MILLISECOND, SECOND, MINUTE, HOUR, DAY, YEAR

The last one is tricky to define. You are required to implement a “Julian year” which assumes we have a leap year every four years, and so each year has one quarter of a leap day as well as the normal 365 days.

Internally a duration is represented in terms of a integral number of milliseconds with a long field (rounding if we have a fraction), but this fact should not be revealed by defining a getter or making the instance fields non-private. Every other class (including Time) using Duration must work with the public methods described above. You are NOT required to handle overflow—if the user tries to define a billion years, that’s their problem.

### 3 Concerning the Time ADT

A value of Time represents a point in historical time, anywhere from the time of the dinosaurs to the distant future. (Strictly speaking, the dinosaurs aren’t historical, which is just was well since the numbers used internally cannot represent time back to the Big Bang. Values of Time should only be used for historical dates and dates in the “near” future: AD 2235 is OK, AD 100000000 is not.) Misuse of the range of time is the fault of the user not the ADT; your code doesn’t need to detect such misuse.

The Time ADT provides the standard operations describe above where toString() returns a UTC (“coordinated universal time” in its French acronym) time string such as “UTC AD 2018/09/04 14:00:00” which is when the first lecture of CS 351 starts. (14:00 is Greenwich Mean Time for 9:00am CDT). You will need to use the library class SimpleDateFormat. Please read online documentation.

The Time ADT provides the following additional operations:

- **Time()** Return the current time (now!).

- **Time(Calendar)** Return the time corresponding to the calendar instance given. Read online documentation for java.util.Calendar.
difference(Time) Return the time between this and the argument as a Duration. The result is always a non-negative duration whether the argument is before or after this time.

add(Duration) Return the new time after adding the given extent of time to this time.

subtract(Duration) Return the new time after subtracting the given extent of time.

You may add additional private constructors (and methods) but no other methods or constructors. Clients of Time must restrict themselves to the public interface.

We recommend that you implement Time using a long field that represents milliseconds after the Java Epoch (UTC AD 1970/01/01 00:00:00). If the number is negative, the time falls before the epoch. As with Duration, the Time class should not reveal the representation, or even the epoch through public methods. The Calendar library class has a method getTimeInMillis which will be useful. Also see the function System.currentTimeMillis() for how to get the current time in terms of milliseconds since the Epoch.

4 Concerning the Period ADT

A period is an extent of historical time. You are not required to make it comparable, but should define the other standard methods, where toString() should return a string of the form [time; duration].

A period has a start time, a length of time and a stop time, but only two of these are necessary to specify a period. You should write three constructors: one omitting the stop, one the length and one the start time.

The Period ADT provides the additional methods:

getStart() Return the time that this period begins.

getStop() Return the time that this period ends.

getLength() Return the length of time of this period.

overlap(Period) Return whether this period overlaps with the argument.

The last method has a simple solution. If your code has more than two conditions, you may be making errors.

5 Test Programs

We provide three JUnit test suites, TestDuration, TestTime and TestPeriod, that you should use to test your ADTs. Do not change them. In Eclipse, you can right click a test and RunAs>JUnit Test.

6 Files

We will be using “git” to access and turn in programs. Follow the instructions given in the lab to clone your homework repository onto your own or lab computer. If you want to get started sooner than that, you can access it from the URI, by SSH from andrew.cs.uwm.edu (port 53211), file path
where *ePanther* is your *login* id.

Make sure you always *commit and push* changes back (*commit* is insufficient) before switching computers and before the deadline. It is a good idea to *commit* (and *push*) regularly while working, so that your progress is preserved even if something goes wrong.

**NEW TO THIS SEMESTER:** You must *commit* your project after each session of work, or when you finish working on a file. This is “showing your work.” Showing your work is essential evidence against any misconduct allegation.

The repository for this assignment includes the following files:

- **src/UnlockTests.java** Unlock all tests without running them. It is *highly* recommended you run this before beginning any implementation.

- **src/TestDuration.java** Unit tests for *Duration*

- **src/TestTime.java** Unit tests for *Time*

- **src/TestPeriod.java** Unit tests for *Period*

- **src/edu/uwm/cs351/Duration.java** “Skeletonized” implementation for *Duration.*

- **src/edu/uwm/cs351/Time.java** Skeleton for *Time*

- **src/edu/uwm/cs351/Period.java** Skeleton for *Period.*

These files must be updated in your *homework1.git* repository before the deadline (10:00 PM, Monday). Homework is due at 10:00 pm, **NOT** midnight!