SimpleTime Laboratory

Overview

In this laboratory, you will program a class called `SimpleTime` that will deal with time measured in hours and minutes. Internally, to simplify computation, time will be represented as the total number of minutes from midnight. You will use computation to calculate the hours between 0 and 23 and the minutes between 0 and 59 from this internal data representation as total minutes.

The primary focus of the laboratory is for you to learn how to build a class, its constructors, set functions, get functions, functions for addition and subtraction, functions for comparisons, and string input-output functions. The `SimpleTime` class is designed to illustrate many patterns that occur over and over in building classes. The member data is as simple as possible, one `int` item called `totalMinutes`. Nevertheless, since this data must be transformed into hours and minutes at certain stages and since there can be problems in dealing with 60 minutes in an hour and 24 hours in a day, this laboratory provides some challenges.

The secondary focus is for you to learn how to thoroughly test a class implementation. We have provided a test program `SimpleTimeTest` that you will work with. A screen snapshot is shown below:
In this snapshot, Time1 has been entered as 700 minutes and Time2 as 900 minutes. The snapshot shows the result after the button “Time1 + Time2” in the rightmost column has been pressed. Since there are 24 * 60 = 1440 minutes in a day, Time1 + Time2 equals 1600 minutes which becomes 160 minutes. This result is shown in the Time Arithmetic Result panel. Similarly, Time1 - Time2 = -200 minutes which becomes 1240 minutes. This result would be shown in the snapshot if the button labeled “Time1 - Time2” had been pressed instead.

If you examine the screen carefully, you will notice that time input may be given in one of three ways: (1) as total minutes; (2) as hours and minutes; and (3) as a text string. In text string input, you may enter the time in any of three formats:

   (a) In 24 hour format (hh:mm with 0 <= hh <= 23 and 0 <= mm <= 59).

   (b) In 12 hour format (hh:mm [AM,PM] with 1 <= hh <= 12; and 0 <= mm <= 59; and a choice of AM or PM)

   (c) As a numeric string representing the total minutes.

Thus, the text strings “00:43”, “12:43 AM”, and “43” are equivalent and correspond to the time that is 43 minutes after midnight. Similarly, “12:43”, “12:43 PM”, and “763” are equivalent and correspond to the time that is 43 minutes after noon. In the same way, “23:43”, “11:43 PM”, and “1423” are equivalent and correspond to the time that is 17 minutes before midnight. Notice, in particular, that the time “12:43 PM” occurs before the time “11:43 PM”. This is why the 12 hour format common in the United States is so peculiar.

When numeric time data is entered in the “Total Minutes” field of the upper input panel or in the “Hours” and “Minutes” fields of the middle input panel, the SimpleTime class should handle large inputs or negative inputs in a consistent mathematical manner. For example, suppose the user enters 2000000 Hours and 1000000 Minutes. Then the SimpleTime class should calculate as follows:

2000000 hours
  = (2000000 % 24) hours
  = 8 hours
  = (8 * 60) minutes
  = 480 minutes.

1000000 minutes
  = (1000000 % 1440) minutes
  = 640 minutes.

Thus, the combined hours and minutes should sum to 480 minutes + 640 minutes = 1120 minutes which is 18:40 or 6:40 PM.

You will be given a helper function, modulus, to assist with these calculations.
The SimpleTime Laboratory Tasks

You task is to provide the missing definitions for the SimpleTime class. This includes the following definitions:

- a: the member data variable totalMinutes
- b: the constructors
- c: the set functions
- d: the get functions
- e: the functions to add and subtract times
- f: the functions to compare times
- e: the functions for string output

Because the function for string input is substantially more difficult than the others, this function will be provided for you. In addition, some technical helper functions will be provided as well.

Note that, in order to compile the test code for this project, you must complete all definitions in (a), (b), and (c), plus the function getTotalMinutes() in (d). Therefore, completing these functions should be your first task. Then, as you finish the remaining definitions in SimpleTime, you may remove certain comments in the test code to enable additional tests. We will describe how to do this later.

The member data variable totalMinutes

This variable totalMinutes should be defined as protected int and initialized to 0. Throughout, the SimpleTime code must maintain the constraint that:

\[ 0 \leq \text{totalMinutes} < 1440 = 24 \times 60. \]

The constructors

You should define the following constructors:

```java
/** Construct the time with the time set to midnight. */
public SimpleTime()

/** Construct the time by using the given minutes as total minutes. */
public SimpleTime(int minutes)

/** Construct the time by using the given hours and minutes. */
public SimpleTime(int hours, int minutes)

/** Construct the time using the given time. */
public SimpleTime(SimpleTime time)
```

The first constructor, the default constructor, is trivial since there is no work to do in the body of the function. The other three constructors should call the corresponding set functions so that the same code is not duplicated. Therefore, these constructors are also almost trivial.
The set functions

You should define the following set functions:

```java
/** Set the time by using the given minutes as total minutes. */
public void setTime(int minutes)
/** Set the time by using the given hours and minutes. */
public void setTime(int hours, int minutes)
/** Set the time using the given time. */
public void setTime(SimpleTime time)
```

The definitions of these functions are not long but they are delicate since you must check for “errors” such as large numbers or negative numbers. Dealing with large numbers can be handled by the remainder operator `%` in the following way. Since there are 24 hours in a day, the value `hours` is equivalent to `hours % 24` as far as time is concerned. Similarly, since there are 1440 minutes in a day, the value `minutes` is equivalent to `minutes % 1440` as far as time is concerned. This reduces large numbers to an appropriate range.

Unfortunately, since the remainder operator `%` returns a negative number if its input is negative, the same simple rules do not work for negative numbers. For example, if we have `-39` hours and we compute `((-39) % 24)` then we obtain `-15` which is still negative.

This is the motivation for defining the following `modulus` function:

```java
/**
 * For radix != 0, returns the unique number n such that
 * 0 <= n < Math.abs(radix) and (value - n) is divisible by radix.
 * Throws ArithmeticException if radix == 0.
 */
public static int modulus(int value, int radix) {
    if (radix == 0)
        throw new ArithmeticException("Error in modulus: radix is 0");

    radix = Math.abs(radix);
    value %= radix;

    if (value < 0)
        value += radix;
    return value;
}
```

If `value % radix` is positive, then `value % radix` and `modulus(value, radix)` are identical. In general, `modulus(value, radix)` is equivalent to `value % radix` as a remainder and is guaranteed to be positive.

The `modulus` function should be used in the function `setTime(minutes)` to force the parameter `minutes` into the desired range `0 <= minutes < 1440 = 24 * 60` before it is stored in the member data `totalMinutes`. 
The function `setTime(hours, minutes)` is more tricky. As in the example above, both `hours` and `minutes` must be reduced using `modulus` to prevent `arithmetic overflow`. Then, the `hours` must be converted to minutes and added to the given `minutes`. Finally, it is necessary to do one further `modulus` operation to prevent overflow in this sum.

The function `setTime(time)` has a different problem of error checking. It is necessary to check whether the parameter `time` is `null` before using its data to set the data `totalMinutes` of this object. Use the following rule: If `time` is `null` then assign `0` to the member data `totalMinutes` otherwise assign the value `time.totalMinutes` taken from the object `time`.

**The get functions**

You should define the following get functions:

```java
/** Return the time expressed as total minutes. */
public int getTotalMinutes()

/** Return the hours component (0 .. 23) of the time. */
public int getHours()

/** Return the minutes component (0 .. 59) of the time. */
public int getMinutes()

/** Return whether the time is AM, that is, before noon. */
public boolean isAM()

/** Return whether the time is PM, that is, equal to or after noon. */
public boolean isPM()
```

Only the first get function directly returns the value of member data. The other functions must perform a computation to return the proper value. It is common in a class for the information to be stored in a minimalist fashion and to calculate when it is necessary to return related information.

**The functions to add and subtract times**

You should define the following arithmetic functions:

```java
/** Add the given minutes to the time. */
public void addTime(int minutes)

/** Add the given hours and minutes to the time. */
public void addTime(int hours, int minutes)

/** Add the given time to the time. */
public void addTime(SimpleTime time)
```
Consider first the pair of functions with the parameter `SimpleTime time`. To perform
the addition or subtraction, it is sufficient to add or subtract `time.totalMinutes` from
the member data `totalMinutes` of the object. Of course, the `modulus` function must be
called to make sure that the resulting value satisfies `0 <= value < 1440`. This may be
done directly or by calling `setTime` to finish the work.

The remaining arithmetic functions can either do their calculations directly or may
create a new temporary `SimpleTime` object `time` based on the incoming parameters. In
the latter case, the new object `time` can then be passed to either `addTime` or
`subtractTime` to finish the work.

Notice that we do not hesitate to let small functions call other small functions to leverage
work already programmed. This makes the design less fragile than it would be if we
constantly repeated calculations. Although there may be a slight performance penalty,
this penalty is usually negligible in the overall running time of an application.

*The functions to compare times*

You should define the following comparison functions:

```java
/** Returns a hash code value for the object. */
public int hashCode() {
  return totalMinutes;
}

/** Indicates whether some other object is "equal to" this one. */
public boolean equals(Object object) {
  if (object instanceof SimpleTime) {
    SimpleTime time = (SimpleTime) object;
    return totalMinutes == time.totalMinutes;
  }
  return false;
}

/** Indicates whether this time is earlier than the given time. */
public boolean isEarlierThan(SimpleTime time)

/** Indicates whether this time is later than the given time. */
public boolean isLaterThan(SimpleTime time)
```

Notice that we have given you the complete code for the first two functions. The reason
for this is as follows. In the Java class `Object` that is the foundation for all other Java
classes, Java defines certain standard methods that may be modified as appropriate by
derived classes. Among these are the pair:
/** Returns a hash code value for the object. */
public int hashCode()

/** Indicates whether some other object is "equal to" this one. */
public boolean equals(Object object)

The `equals` method determines whether the given object should be treated as equal to the other object based on the type and internal data of both objects. By default, this method returns true only if the objects are identical. For `SimpleTime` objects, however, we want to consider the objects as the same if they contain the same internal data. Thus, we override Java's default definition of `equals` to return true if the parameter `object` is in fact a `SimpleTime` object and if its `totalMinutes` data is the same as that of the given object.

By an important Java convention, two entities that test as true for `equals` must return the same value by a special function called `hashCode`. This function is used to insert objects into special tables called hash tables. Be that as it may, once we have modified the `equals` method, we are required to modify the `hashCode` method accordingly and this is what we have done.

Your work here is to copy the definitions of `equals` and `hashCode` into the file and to provide the definitions of `isEarlierThan` and `isLaterThan`.

### The functions for string output

You should define the following functions for string output of `SimpleTime` data:

```java
/** Return the time as total minutes. */
public String toTotalMinutes()

/** Return the time in 24 hour format. */
public String to24HourFormat()

/** Return the time in 12 hour format with AM or PM. */
public String to12HourFormat()

/** The Java toString() method. */
public String toString()
```

To create these functions, you need to know how Java builds `String` data. To create a `String`, you can concatenate information using the `+` sign. As long as at least one item in the concatenation is a `String` the entire expression will be converted to `String`. If there is an ambiguity, you should parenthesize so Java can understand what you intend. The simplest case of the method of concatenation is to convert a `number` to a `String`. Simply use the expression:

```
"" + number
```

Since the first term is a `String` (albeit with zero characters), the number is converted to a `String`. The method of concatenation can be used to create arbitrarily complex `String` objects from constant `String`s, from functions that return `String`s, and from pure data.
The first three functions that you must build name the **String** output by the type of format to be used. The first function `toTotalMinutes` simply converts the member data to a **String** using the method of concatenation. The second and third functions need to extract hour and minute data, insert a colon, and possibly add " AM" or " PM".

We cannot write a treatise here on time measurement so let us simply give a table with some examples of data represented in the three formats:

<table>
<thead>
<tr>
<th>Total</th>
<th>24 Hour</th>
<th>12 Hour</th>
<th>Total</th>
<th>24 Hour</th>
<th>12 Hour</th>
<th>Total</th>
<th>24 Hour</th>
<th>12 Hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00:00</td>
<td>12:00 AM</td>
<td>599</td>
<td>09:59</td>
<td>9:59 AM</td>
<td>781</td>
<td>13:01</td>
<td>1:01 PM</td>
</tr>
<tr>
<td>61</td>
<td>01:01</td>
<td>1:01 AM</td>
<td>719</td>
<td>11:59</td>
<td>11:59 AM</td>
<td>1379</td>
<td>22:59</td>
<td>10:59 PM</td>
</tr>
<tr>
<td>195</td>
<td>03:15</td>
<td>3:15 AM</td>
<td>720</td>
<td>12:00</td>
<td>12:00 PM</td>
<td>1439</td>
<td>23:59</td>
<td>11:59 PM</td>
</tr>
</tbody>
</table>

To extract hour and minute data, you should of course use the get functions. To format this data, we have provided a rather specialized helper function:

```java
/**
 * If the given value is between 0 and 99
 * return a String representing the value as two digits
 * otherwise return the error String "XX".
 */
protected String twoDigitString(int value)
```

This function will produce the two digit **String** values for the hours and minutes that need a leading 0 character. You may want to read this function to see how it is built using the method of concatenation.

The final function that you must build is the Java `toString` method. The `toString` method is what Java calls to implement the method of concatenation. When an object appears in a concatenation sequence, Java calls its `toString` method to get a **String** representation of its information as far as is possible. You can make the `toString()` method return the same value as the `to24HourFormat()` method.

**The Concept of Stringable**

The Java Power Tools define the notion of a **Stringable** object as one whose state can be fully encapsulated into a **String** and then faithfully extracted from a **String** that has the suitable format. The `toStringData` method performs the encapsulation into a **String** and the `fromStringData` method performs the extraction of data from the **String** into the object. We have decided to provide you with both of these methods to simplify your work.
Testing Your Implementation of SimpleTime

The test program SimpleTimeTest is already provided. However, it is written assuming that certain methods in SimpleTime have been implemented. These are the methods listed above in (a), (b), and (c) plus the getTotalMinutes() method. You should start testing as soon as these methods have been written.

In the test program, we have placed comments in certain locations. Once you remove these comments, additional tests will be performed.

The first set of comments is in the file TimeStatsView.java in the refresh() function:

```java
protected void refresh() {
    // stats[0].setText("" + model.getTotalMinutes());
    // stats[1].setText("" + model.getHours());
    // stats[2].setText("" + model.getMinutes());
    // stats[3].setText("" + model.isAM());
    // stats[4].setText("" + model.isPM());
    // stats[5].setText(model.toTotalMinutes());
    // stats[6].setText(model.to12HourFormat());
    // stats[7].setText(model.to24HourFormat());
}
```

As you can see, you can remove the comments one-by-one as you define the methods being called.

The other set of comments is in the file SimpleTimeTest.java:

```java
protected void addTimeTest() {
    /*
    SimpleTime time0 = output[0].getTime();
    SimpleTime time1 = output[1].getTime();
    SimpleTime time2 = new SimpleTime(time0);
    time2.addTime(time1);
    output[2].setTime(time2);
    */
}

protected void subtractTimeTest() {
    /*
    SimpleTime time0 = output[0].getTime();
    SimpleTime time1 = output[1].getTime();
    SimpleTime time2 = new SimpleTime(time0);
    time2.subtractTime(time1);
    output[2].setTime(time2);
    */
}
```

These comments should be removed once the addition and subtraction methods have been defined.

It is a very common practice in industry to write the test code for a class before the class is written or in parallel with the writing of the class. This helps to guarantee that the test code is not biased by the actual code of the class. If you look very carefully at our test program, you will find a few methods in SimpleTime that you are write which are not tested. This is a slight flaw in the test program but it makes the user interface more suitable for a laboratory exercise.