# Java By Abstraction - Test-D (Chapters 7-12)

| | |
|---|---|
| **Last Name** | |
| **First Name** | |

Do not write below this line

| | |
|---|---|
| Q1 (35%) | |
| Q2 (35%) | |
| Q3 (30%) | |
| TOTAL | |

| **String Methods (invoke on a string s)** | `char` **`charAt(int p)`**<br>Returns the character at position# p in s. |
|---|---|
| `int` **`compareTo(String t)`**<br>Returns a negative number if s<t, zero if s=t, and a positive number if s>t. | **Static Methods** |
| `int` **`indexOf(String t, int f)`**<br>Looks for the string t within s, starting at position# f in s. Returns the position in s where the match was found. Returns -1 if no match was found. | `Integer.`**`parseInt`**`(s)`<br>`Double.`**`parseDouble`**`(s)`<br>`methods to convert a string` **`s`** `that contains a number to a primitive type.` |
| `int` **`indexOf(String t)`**<br>Looks for the string t within s (as above), starting at the beginning of s. | `double` **`abs(double x)`**<br>Returns the absolute value of x (`Math`) |
| `String` **`substring(int f,int t)`**<br>Returns all characters in s with position numbers ≥ f and < t. | `double` **`pow(double x, double y)`**<br>Returns x raised to y. (`Math`) |
| `String` **`substring(int f)`**<br>Returns a substring of s that begins at f and extends to the end of s. | `double` **`rint`**`(double a)`<br>Returns the closest `double` to a that is equal to a mathematical integer. (`Math`) |
| `String` **`replace(char x,char y)`**<br>Returns a string with all occurrences of character x in s replaced by y. | `String` **`repeat(int count,char c)`**<br>Returns a string made up of c repeated `count` times. (`IO`) |
| `String` **`toUpper/LowerCase()`**<br>Returns a string of all characters in s converted to upper / lower case. | **`String Tokenizer`**<br>**`Constructor:`** `(String s, String delimiters)`<br>**`Methods:`** `String nextToken(), boolean hasMoreTokens(), int countTokens()` |
| `String` **`trim()`**<br>Returns the same content as s but with any leading/trailing white-space | |

## QUESTION 1 *<35 points>*

Consider the classes `Route`, `Time`, `Trip`, `XTrip`, and `Log` whose API appear at the end of this booklet. One of the transit routes is route number 11 whose lower station is "Union" and whose upper station is "Steeles". On average, a regular trip along this route takes 105 minutes, but an express one takes only 45 minutes.

Write an app that prompts the user to enter a route number and then generates a report that lists all trips along that route in the log returned by the `getRandom` method. This method returns a collection of randomly chosen trips along a variety of routes. Your app must extract from it only those trips that are along the route entered by the user, and then generate a report that looks like this (assuming the user entered route number 11):

```
Route Number 11, Union <--> Steeles (105 min)

From        Vehicle     Departed    Arrived     Deviation
Union       700         08:30       10:30       15
Union       400         08:45       10:30       0
Steeles     700         10:40       12:10       -5
Union       512-X       10:45       11:35       5
Steeles     400         10:50       12:30       -5
```

The report starts with a line that identifies the route number, the route's lower / upper stations, and the expected travel time between them. Then, a line is generated for each trip in the log showing where it started, the vehicle number, the departure / arrival times, and the deviation from the schedule.

The first row indicates that vehicle 700 left Union at 08:30 and arrived at Steeles at 10:30. This means it took 120 minutes and that is why the 5th column shows 15, the difference between the actual duration (120 min) and the theoretical one (assumed in the above example to be 105 min). The fourth row depicts an express trip. It adds a "`-X`" suffix to the vehicle number and takes the express nature into account when it computes the deviation. For this trip, the `getSavedTime` method returns 65, and hence, the expected time for it is 105 - 65 = 45 min. That is why the last column shows a deviation of 5 min between the actual time (50) and the express time (45).

The above output is only an example but your code must reproduce the above layout (the entries in the table are all tab-delimited) for any log and any route. You can assume, however, that there is at least one trip in the log along the user-entered route; i.e. the report has at least one row.

Note that the ordering of lines in the report must follow that of the log; i.e. you do not have to order the trips in any way, just output them in the order you retrieve them from the log.

Write your answer on the next page.

```
import type.lang.*;

public class Transit
{  public static void main(String[] ar)
   {
```

**QUESTION 2** *<35 points >*

We want to develop a billing app for a cellular phone company. The app reads a file called "`calls.txt`" that contains information about phone calls made by a customer and generates a bill for the customer to pay. Each record in the file represents a phone call and consists of three pipe-delimited fields: `rate|start|end`

The first field is the charge rate for the phone call and is expressed in cents per second. The second field is the time at which the call started, and the third is the time at which the call ended. For example, the record:

```
0.15|Sep 1, 2003 11:10:56 PM|September 2, 2003 1:17:40 am
```

represents a phone call that started at 11:10:56 pm on September 1st and ended at 1:17:40 am on the following day. Hence, this call lasted for 2 hours, 6 minutes, and 44 seconds, or 7,604 seconds. At 0.15 cent/sec, this call leads to a charge of: 0.15 * 7604 / 100 = $11.41.

One of the challenges facing this app is the variability of the format in which the start / end times are expressed, e.g. the month name is sometimes abbreviated. To overcome this, the app delegates the date/time extraction to the `parse` method of the `DateFormat` class. This class does not have a constructor but does have a `static` method that returns an instance of it. Once the string is converted to a `Date`, we can use the `getTime` method of `Date` to convert it to `long` (measured in milliseconds). The relevant parts of the API of both classes are reproduced below.

**java.text**

## Class DateFormat

| Method Summary | |
|---|---|
| `static DateFormat` | `getDateTimeInstance()`<br>Gets the date/time formatter with the default formatting style for the default locale |
| `Date` | `parse(String source)`<br>Parses text from the given string to produce a date.<br>**Throws:** `java.text.ParseException`<br>if the specified string cannot be parsed. |

**java.util**

## Class Date

| Method Summary | |
|---|---|
| `long` | `getTime()`<br>Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object. |

The app must implement the following algorithm:

1. Open the file `calls.txt`
2. Read its records one by one
3. Extract the `rate` and turn it into a `double`
4. Extract the `start` date/time and turn it into a `long`
5. Extract the `end` date/time and turn it into a `long`
6. Compute the charge in dollars using `(rate/100)*(end - start)/1000`
7. Output the number of the processed record (counting starts at zero), followed by a tab, and followed by the charge.
8. Process the next record.

The app must use Java's exception handling. If an error were encountered in a record, one of the following error messages would replace the charge in the output (step #7 above):

| Error Message | When to issue |
|---|---|
| `Non-numeric rate!` | If `rate` cannot be parsed into a `double` |
| `"start" is not a date!` | If `start` cannot be parsed into a `Date` |
| `"end" is not a date!` | If `end` cannot be parsed into a `Date` |
| `Missing field!` | If one of the fields is missing |
| `"start" is later than "end"!` | If `start` occurs later in time than `end` |

Here is a sample file:

```
0.15|Sep 1, 2003 11:10:56 PM|September 2, 2003 1:17:40 am
0.75|September 5, 2003 2:17:34 pm|Sep 5, 2003 2:30:0 pm
0.75|September 5, 2003 2:17:34 pm
$1.03|September 5, 2003 2:17:34 pm|Sep 5, 2003 2:30:0 pm
1.03|Sept. 5, 2003 2:17:34 pm|Sep 5, 2003 2:30:0 pm
1.03|September 5, 2003 2:17:34 pm|5/9/2003 2:30:0 pm
0.75|September 5, 2003 2:17:34 pm|Sep 3, 2003 2:30:0 pm
```

And here is its corresponding sample run:

```
0    OK   $11.41
1    OK   $5.60
2    Missing field!
3    Non-numeric rate!
4    "start" is not a date!
5    "end" is not a date!
6    "start" is later than "end"!
```

Recall that `StringTokenizer` throws a `java.util.NoSuchElementException` when a token is missing and that `Double` throws a `java.lang.NumberFormatException` when a non-numeric character is encountered.

```
import type.lang.*;
import java.util.*;
import java.text.*;

public class Bill
{  public static void main(String[] ar)
   {
```

## QUESTION 3 *<30 points >*

Write an app that reads a file of tokens, one token per record, and produces an ordered table whose rows correspond to the word lengths found in the file. For example, if the input file were as shown on the right then the produced output would be as shown below:

```
the
word
arm
a
quite
well
a
the
point
end
aaaaaaaa
arrive
try
indent
k
exit
echo
far
```

```
1    [a, k]
3    [arm, end, far, the, try]
4    [echo, exit, well, word]
5    [point, quite]
6    [arrive, indent]
8    [aaaaaaaa]
```

The first row, for example, corresponds to words of length 1, and hence, it contains 1 followed by a tab, followed by the set of all words of length 1. Notice that each set is sorted alphabetically and that the rows are sorted in increasing length order. Note also that duplicates are removed. The word "`the`", for example, appears twice in the table but only once in the output. Note that the tokens in the files should be treated as strings of arbitrary lengths and contents; i.e. do not assume they must be English words.

*Hint:*
*Create a map whose key is the length of a word in the file and whose value is a set of all words in the file having that length. Whenever a word is read, the map is searched to see if it has an element with that length as key. If so, the new word is added to the element's set. Otherwise, a new element is created in the map.*

Here is some relevant API from the collection's framework:

| SET | MAP |
|---|---|
| *To add an element:* | |
| `boolean`<br>**`add(Object e)`** | `Object`<br>**`put(Object key, Object value)`** |
| *To access an element:* | |
| *none* | `Object`<br>**`get(Object key)`** |
| *To search for an element:* | |
| `boolean`<br>**`contains (Object o)`** | `boolean`<br>**`containskey (Object key)`** |
| *To traverse all elements:* | |
| `Iterator:` **`iterator()`**<br>*invoke on it:*<br>`Object` **`next()`**<br>`boolean` **`hasNext()`** | `Iterator:` **`keySet().iterator()`**<br>*invoke on it:*<br>`Object` **`next()`**<br>`boolean` **`hasNext()`** |

```
import type.lang.*;
import java.util.*;

public class LengthMode
{     public static void main(String[] ar)
      {
```

- ## Class Route

This class encapsulates a public transit route. Each route is identified uniquely by its *route number* (an integer in [1,999]) and the class uses this number as a key to a central database in order to retrieve the names of the two stations that the route connects (known as the *lower* and the *upper* stations) and the average travel time between them.

### Constructor Summary

`Route(int routeNumber)`
Construct a Route object having the passed route number. The route data is retrieved from a central database based on the passed `routeNumber`. If the passed number is invalid (less than 1 or greater than 999) or if it is not found in the database, a precondition exception will be thrown.

### Method Summary

| | |
|---|---|
| `int` | `getRouteNumber()`<br>Return the number of this route. |
| `String` | `getLowerStation()`<br>Return the name of one end of this route. |
| `String` | `getUpperStation()`<br>Return the name of the other end of this route. |
| `int` | `getExpectedTime()`<br>Return the time, in minutes, that it takes a transit vehicle on average to travel from either end of this route to the other. |

- ## Class Time

This class encapsulates the time-of-day, in hours and minutes, using a 24-hour clock.

### Constructor Summary

`Time(int hh, int mm)`
Construct a `Time` object. A precondition exception will be thrown if the hour `hh` is not in [0, 23] or if the minute `mm` is not in [0, 59].

### Method Summary

| | |
|---|---|
| `void` | `add(int m)`<br>Add `m` minutes to this time instance. |
| `int` | `getInterval(Time other)`<br>Return the absolute value of the shortest number of minutes between this time and `other`, e.g. if one time is 10:15 and the other is 10:00, the return would be 15. And if one time is 23:45 and the other is 00:15, the return would be 30. |
| `String` | `toString()`<br>Return the 5-character string `hh:mm` (hh and mm are zero-filled; e.g. 03:05) |

- **Class Trip**

This class encapsulates a trip of a particular transit vehicle along a route. It holds the vehicle number, the route of the trip, the direction along the route, and the departure/arrival times.

## Field Summary

| | |
|---|---|
| `static final int` | `UP`<br>This value indicates that the trip begins at the *lower* station of the route and ends at its *upper* station. |
| `static final int` | `DOWN`<br>This value indicates that the trip begins at the *upper* station of the route and ends at its *lower* station. |

## Constructor Summary

`Trip(int vehicle, Route r, int direction, Time start, Time end)`
Construct a `Trip` that starts at time `start` and ends at `end` along route `r`. The `Trip` starts at the lower station of the route and ends at its upper station if `direction` is `UP` (a class constant); otherwise, it is in the opposite direction. A precondition exception will be thrown if `direction` is neither `UP` nor `DOWN;` the `start` time is not earlier than the `end` time; or the vehicle number, `vehicle`, is not positive.

## Method Summary

| | |
|---|---|
| `int` | `getVehicleNumber()`<br>Return the number of the vehicle of this `Trip`. |
| `Route` | `getRoute()`<br>Return the route of this `Trip`. |
| `int` | `getDirection()`<br>Return the direction of this `Trip`. |
| `Time` | `getStartTime()`<br>Return the start time of this `Trip`. |
| `Time` | `getEndTime()`<br>Return the end time of this `Trip`. |
| `String` | `toString()`<br>Return the string: "`A Transit Trip`". |
| `boolean` | `equals(Object other)`<br>Return `true` if `other` is a `Trip` instance having the same route number, direction, start time, and end time as this trip. The return is `false` otherwise. |

- ## Class Log

This class encapsulates a collection of trips and provides methods for traversing the stored trips per route and in non-descending start time order.

| Constructor Summary |
| --- |
| `Log()`<br>Construct an empty log capable of holding an arbitrary number of `Trip` objects. |

| Method Summary | |
| --- | --- |
| `void` | `add(Trip t)`<br>Add `t` to this log. |
| `Trip` | `getFirst(int routeNumber)`<br>Return a trip from this log having the passed route number and the earliest start time. The return is `null` if no trips along the passed route exist in this log. |
| `Trip` | `getNext()`<br>Return a trip from this log having the route number indicated in the last invocation of `getFirst` and the next earliest start time. The return is `null` if no more trips along the passed route exist in this log. |
| `static Log` | `getRandom()`<br>Return a reference to a randomly created log. |

- ## Class XTrip extends Trip

This class encapsulates an *express* trip of a public transit vehicle along a route. The vehicle in such a trip does not stop in any intermediate station as it travels from one end of the route to the other.

| Constructor Summary |
| --- |
| `XTrip(int vehicle, Route r, int direction, Time start, Time end)`<br>Exactly the same behavior as the superclass constructor. |

| Method Summary | |
| --- | --- |
| `int` | `getSavedTime()`<br>Return the time, in minutes, that is saved on average due to the express nature of this trip. |
| `String` | `toString()`<br>Return the string: "`An Express Transit Trip`". |
| `boolean` | `equals(XTrip other)`<br>Return `true` if `other` has the same route number, start time and end time as this trip (regardless of direction). The return is `false` otherwise. |