Java By Abstraction - Test-C (Chapters 7-9)

Last Name	
First Name	

Do not write below this line

												Q1 (30%)	
									Q2 (30%)				
									Q3A (10%)				
Q3B (30%)													
												TOTAL	

String Methods (invoke on a string s)	char **charAt(int p)** Returns the character at position# p in s.
int **compareTo(String t)** 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)** 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) Double.**parseDouble**(s) methods to convert a string **s** that contains a number to a primitive type.
int **indexOf(String t)** Looks for the string t within s (as above), starting at the beginning of s.	double **abs(double x)** Returns the absolute value of x (Math)
String **substring(int f,int t)** Returns all characters in s with position numbers ≥ f and < t.	double **pow(double x, double y)** Returns x raised to y. (Math)
String **substring(int f)** Returns a substring of s that begins at f and extends to the end of s.	double **rint**(double a) Returns the closest double to a that is equal to a mathematical integer. (Math)
String **replace(char x,char y)** Returns a string with all occurrences of character x in s replaced by y.	String **repeat(int count,char c)** Returns a string made up of c repeated count times. (IO)
String **toUpper/LowerCase()** Returns a string of all characters in s converted to upper / lower case.	**String Tokenizer** **Constructor**: (String s, String delimiters) **Methods**: String nextToken(), boolean hasMoreTokens(), int countTokens()
String **trim()** Returns the same content as s but with any leading/trailing white-space	

QUESTION 1 *<30 points >*

Write the app `CodeStyle` that reads a text file containing a syntactically correct Java program and generates a second text file containing the same program except its whitespace content is formatted as per our coding style. Here is a sample run:

```
java CodeStyle
Filename: Test.txt
Done. The file: "StyledTest.txt" has been generated.
```

The app reads the name of the input filename from the user and then generates the output on a file having the same name but with the prefix: "`Styled`". Here are the contents of the two files:

File: Test.txt

```
import type.lang.*;
     import     type.lib.*;

public     class Example
{  public static void        main(String[] args)
 {      for (int i = 0; i < 5; i++)
           {    for (int j =    0;   j < 3;    j++)
   {  IO.print(i);
IO.print(" ");
                IO.println(j);
}
     }
   }
}
```

File: StyledTest.txt

```
import type.lang.*;
import type.lib.*;

public class Example
{	public static void main(String[] args)
	{	for (int i = 0; i < 5; i++)
		{	for (int j = 0; j < 3; j++)
			{	IO.print(i);
				IO.print(" ");
				IO.println(j);
			}
		}
	}
}
```

Specifically, your app must: (1) read the file line by line and remove any leading whitespace from its lines; (2) remove any extra whitespace within the line and replace it by a single space (or a single tab if following an open brace); and (3) indent each block by one tab relative to its braces. You can assume that any open / closed brace in the original file appears as the first non-whitespace character in its line, and that it is separated from any following token by white space.

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

public class CodeStyle
{  public static void main(String[] ar)
   {  IO.print("Filename: ");
      String inFile = IO.readLine();
      String outFile = "Styled" + inFile;
```

QUESTION 2A *<15 points >*

Pages 10 and 11 of this booklet depict the API of three classes: `Route`, `Time`, and `Trip`. One of the transit routes is route number 11 whose lower station is "Union" and whose upper station is "Steeles". Let us assume that transit vehicle number 700 made a trip up this route. It left Union at 11:30 am and arrived at Steeles 1 hour and 45 minutes later. In order to represent this trip, an object was created as follows:

```
      Trip t = new Trip("700", 11, UP, "11:30", "1:15");
```

Each of the following 5 questions (at 4 points each) deals with one of the parameters in the above constructor. For each, either state that the parameter is correct as written or correct it.

- *First parameter:* `"700"`

- *Second parameter:* `11`

- *Third parameter:* `UP`

- *Fourth parameter:* `"11:30"`

- *Fifth parameter:* `"1:15"`

QUESTION 2B *<15 points >*

Assume that the object in question 2A was correctly created and that `t` is a reference to it. Given `t`, consider how it is used in the following 5 questions. For each, if you think the usage is incorrect, write "incorrect" and explain the reason briefly. Otherwise, write the output.

- `IO.println(t.getRouteNumber());`

- `IO.println(t.getRoute().getUpperStation());`

- `IO.println(t.getStartTime());`

- `IO.println(t.Route.getLowerStation());`

- `IO.println(t.getEndTime().add(10));`

QUESTION 3A *<10 points >*

`XTrip` is a subclass of `Trip` that represents express trips, ones that don't stop in between the two route ends. Its API is shown at the end of this booklet.

3A.1 Classify all the methods in `XTrip`. For each, state whether it is new, inherited, overriding, or cross-class overloading.

3A.2 Draw a UML diagram that depicts the relationship between the 5 classes: `Route`, `Time`, `Trip`, and `XTrip`.

QUESTION 3B *<30 points >*

Consider an app that deals with route number 11 whose lower station is Union and upper station is Steeles. On average, a regular trip along this route takes 105 minutes, but an express one takes only 45 minutes. The app starts by the declarations:

```
Trip t1;
Trip t2;
Trip t3;
XTrip t4;
```

3B.1 Write one or more statements to create a `Trip` for vehicle number 20 that leaves Union at 10:05 and arrives at Steeles at 11:40. Make `t1` a reference to it.

3B.2 Write one or more statements to create an `XTrip` for vehicle number 20 that leaves Union at 10:05 and arrives at Steeles at 11:40. Make `t2` a reference to it.

3B.3 Write one or more statements to create an `XTrip` for vehicle number 20 that leaves Steeles at 10:05 and arrives at Union at 11:40. Make `t3` a reference to it.

3B.4 Write one or more statements to create an `XTrip` for vehicle number 20 that leaves Steeles at 11:45 and arrives at Union at 13:40. Make `t4` a reference to it.

`t1`, `t2`, `t3`, and `t4` are now properly declared and initialized. In each of the following parts, we are going to add one output statement to the app, and you need to state its output and justify your answer (i.e. write a brief explanation of how you reached your conclusion). If you think the added statement will produce syntax or a runtime error, state so and also justify your answer. Treat the parts independently.

3B.5 `IO.println(t1);`

3B.6 `IO.println(t3);`

3B.7 `IO.println(t4);`

3B.8 `IO.println(t3.getSavedTime());`

3B.9 `IO.println(t1.equals(t2));`

3B.10 `IO.println(t2.equals(t3));`

3B.11 `IO.println(((XTrip) t2).equals(t3));`

3B.12 `IO.println(((XTrip) t2).equals((Xtrip) t3));`

- ## 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()` Return the number of this route.
`String`	`getLowerStation()` Return the name of one end of this route.
`String`	`getUpperStation()` Return the name of the other end of this route.
`int`	`getExpectedTime()` 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)` Add `m` minutes to this time instance.
`int`	`getInterval(Time other)` 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()` 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` This value indicates that the trip begins at the *lower* station of the route and ends at its *upper* station.
`static final int`	`DOWN` 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()` Return the number of the vehicle of this `Trip`.
`Route`	`getRoute()` Return the route of this `Trip`.
`int`	`getDirection()` Return the direction of this `Trip`.
`Time`	`getStartTime()` Return the start time of this `Trip`.
`Time`	`getEndTime()` Return the end time of this `Trip`.
`String`	`toString()` Return the string: "`A Transit Trip`".
`boolean`	`equals(Object other)` 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()` Construct an empty log capable of holding an arbitrary number of `Trip` objects.

Method Summary	
`void`	`add(Trip t)` Add `t` to this log.
`Trip`	`getFirst(int routeNumber)` 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()` 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()` 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)` Exactly the same behavior as the superclass constructor.

Method Summary	
`int`	`getSavedTime()` Return the time, in minutes, that is saved on average due to the express nature of this trip.
`String`	`toString()` Return the string: "`An Express Transit Trip`".
`boolean`	`equals(XTrip other)` 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.