Timestamp: 2020/09/04 15:23

CSC1016S Assignment 3

Class declarations with Fields, Constructors, Methods and Access Modifiers

Introduction

This assignment involves building, testing and debugging Java programs, that create and manipulate composition of classes that model simple types of object; and that employ the full gamut of implementing technologies: fields, constructors, methods and access modifiers.

The theme remains as in assignment 2. the simulation of a pay-to-stay car park.

The kind of car park in question has a ticket machine at the entrance and a cashier at the exit. A driver, on entering the car park receives a ticket stamped with the arrival time. (The arrival time is also recorded on the magnetic strip on the back.) On exit, the driver gives the ticket to the cashier, the duration of the stay is calculated and from that, how much must be paid.

In assignment 2 you familiarised yourself with the Time, Duration, Money and Currency classes, and constructed Ticket, and Register classes. The ultimate aim is to build a complete pay-to-stay simulation.

Here is sample I/O from the final program:

```
Car Park Simulator
The current time is 00:00:00.
Commands: tariffs, advance {minutes}, arrive, depart, quit.
>tariffs
[0 minutes .. 30 minutes] : R10.00
[30 minutes .. 1 hour] : R15.00
[1 hour .. 3 hours] : R20.00
[3 hours .. 4 hours] : R30.00
[4 hours .. 5 hours] : R40.00
[5 hours .. 6 hours] : R50.00
[6 hours .. 8 hours] : R60.00
[8 hours .. 10 hours] : R70.00
[10 hours .. 12 hours] : R90.00
[12 hours .. 1 day] : R100.00
>arrive
Ticket issued: Ticket[id=80000001, time=00:00:00].
>advance 20
The current time is 00:20:00.
>arrive
Ticket issued: Ticket[id=80000002, time=00:20:00].
>advance 15
The current time is 00:35:00.
>depart 80000001
Ticket details: Ticket[id=80000001, time=00:00:00].
Current time: 00:35:00.
Duration of stay: 35 minutes.
Cost of stay : R15.00.
>advance 6
The current time is 00:41:00.
```

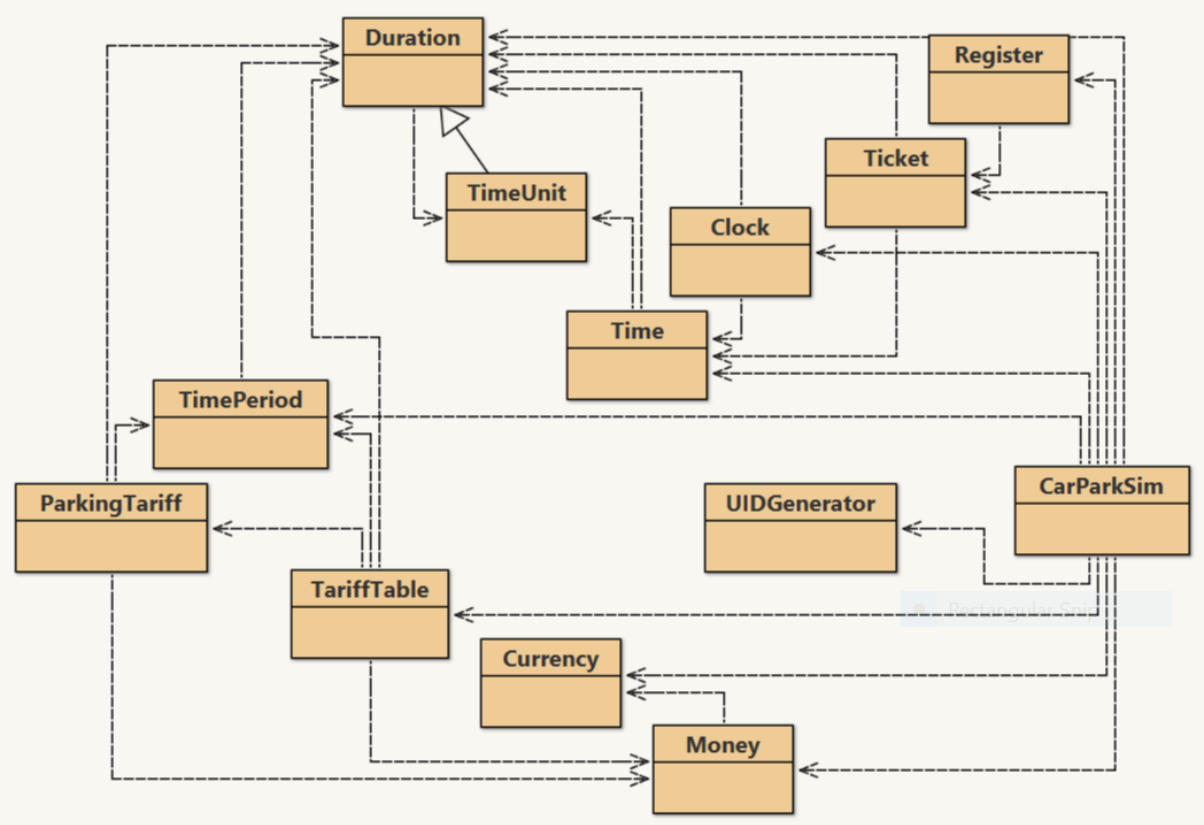
CONTINUED

```
>depart 80000002
Ticket details: Ticket[id=80000002, time=00:20:00].
Current time: 00:41:00.
Duration of stay: 21 minutes.
Cost of stay : R10.00.
>quit
Goodbye.
```

Items in bold represent user input.

- There's a 'tariff' command that prints a list of parking tariffs.
- The "arrive" command is used to record the arrival of a vehicle and causes a ticket to be issued.
- The "depart" command is used to record the departure of a vehicle and causes the duration of the stay to be calculated. The cost of stay is printed after the duration.
- Time is simulated. Initially it is midnight (00:00:00). The "advance" command is used to advance the current time by a given number of minutes.

The following diagram depicts the classes that form the program:



An arrow from a class A to a class B indicates that A uses B in some way.

A Ticket object represents a car park ticket. It has a unique ID and time of issue (24-hour clock).

The job of a Register object in the simulation is to store all the tickets that have been issued. When a Ticket is issued, it is stored in the register. When the driver departs, the ticket ID is used to retrieve the Ticket object to calculate the duration of stay.

CONTINUED

A Clock object is used to simulate time and the passing of time. Basically, it stores a time value that can be advanced. It may be found on the Vula page for this assignment.

The Time and Duration classes are essentially the same as used in assignment 1. We have tweaked them a bit and have created a new TimeUnit class. You should use these versions for this assignment. (The big change is the addition of some string formatting for Duration.)

The UIDGenerator class is used to generate unique IDs for Ticket objects. It may be found on the Vula page for this assignment.

A TimePeriod is a duration range. It has an inclusive lower bound, l, and an exclusive upper bound, u. A time duration, d, falls within the range if $l \leq d < u$.

For example, a TimePeriod might be 30 up to (but not including) 90 minutes, where 30 is the inclusive lower bound, and 90 is the exclusive upper bound. A duration of 90 minutes does not fall within the period, a duration of 89 minutes does.

A ParkingTariff represents the parking tariff, t, for stays that fall within a given time period, p.

For example, if you park your car in the Cavendish Square shopping mall (in Claremont, Cape Town), stays from 2 hours up to (but not including) 3 hours cost R15.

A TariffTable represents a series of parking tariffs in ascending order of time period.

For example, (taken from Cavendish Square again)

Time Period	*Tariff*
0 – 2 Hours	R5
2 – 3 Hours	R10
3 – 4 Hours	R20
4 – 5 Hours	R30
5 – 6 Hours	R50
6 – 7 Hours	R70

The CarParkSim class contains the main program method. It creates the TariffTable, Register and Clock objects and handles user input/output.

- When the user enters the 'arrive' command it creates a Ticket object, prints it and stores it in the register.
- When the 'depart *<id>*' command is entered, the given ticket ID is used to retrieve the relevant Ticket object from the Register. The current time is obtained from the Clock and the duration of stay calculated then printed.
- When 'advance *<minutes>*' is entered, the time stored by the Clock object is advanced by the given amount.

The program may seem a little daunting a first glance, but we'll take it one step at a time.

Completion of a cut-down version of the simulator is the subject of exercise one. We'll ignore the issue of payment, and use Clock, Ticket and Register to create a simulation that allows time to advance, tickets to be issued and duration of stay calculated.

Having built a basic simulation, we'll then add payment in.

- Construction of a TimePeriod class is the subject of exercise 2.
- Construction of a TariffTable class is the subject of exercise 4 and requires the design and construction of a ParkingTariff class, which is the subject of exercise 3.

Revising the CarParkSim class to include the new classes will be deferred until the next assignment (though, since you know where it's headed, you could finish it before then.)

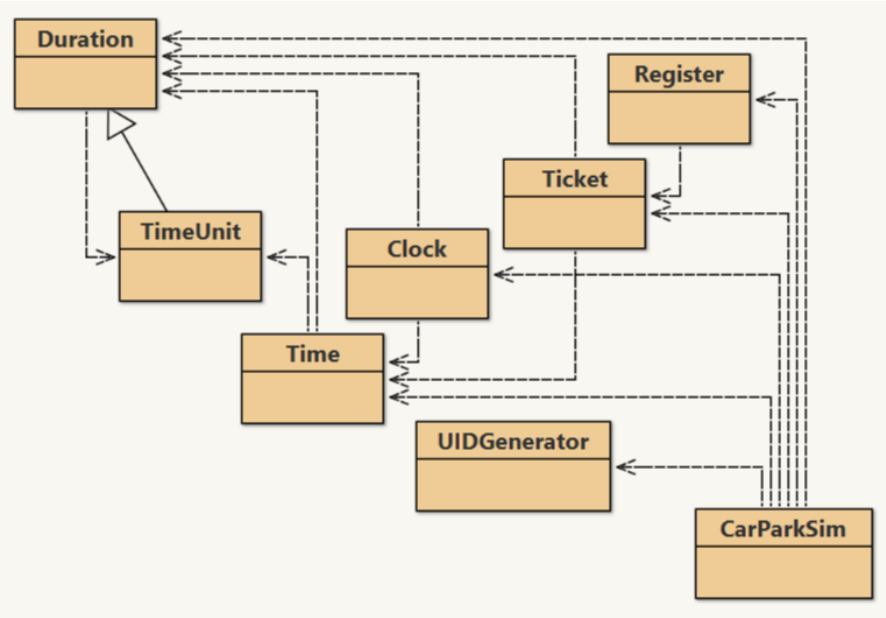
For this assignment we expect to see the use of access modifiers to properly implement classes. In this regard, class specifications are properly abstract – they describe a type of object's attributes and behaviour but not how it's implemented i.e. there are no details of instance variables and how they are manipulated.

The assignment will be part automatically marked and part manually marked. The tutors will review your implementation decisions; your choice of instance variables and data types.

Exercise 1 [25 marks]

On the assignment page you will find revised Time and Duration classes, the Clock TimeUnit and UIDGenerator classes, and a skeleton implementation of the CarParkSimulator class – your task is to fill in the blanks.

Here is the design:



And here is sample I/O.

```
Car Park Simulator
The current time is 00:00:00.
Commands: advance {minutes}, arrive, depart, quit.
```

CONTINUED

```
>arrive
Ticket issued: Ticket[id=80000001, time=00:00:00].
>advance 1
The current time is 00:01:00.
>arrive
Ticket issued: Ticket[id=80000002, time=00:01:00].
>advance 15
The current time is 00:16:00.
>depart 80000001
Ticket details: Ticket[id=80000001, time=00:00:00].
Current time: 00:16:00.
Duration of stay: 16 minutes.
>advance 6
The current time is 00:22:00.
>depart 80000002
Ticket details: Ticket[id=80000002, time=00:01:00].
Current time: 00:22:00.
Duration of stay: 21 minutes.
>quit
Goodbye.
```

Items in bold represent user input. The program accepts a series of commands.

- The “arrive” command is used to record the arrival of a vehicle and causes a ticket to be issued.
- The “depart” command is used to record the departure of a vehicle and causes the duration of the stay to be calculated.
- Time is simulated. Initially it is midnight (00:00:00). The “advance” command is used to advance the current time by a given number of minutes.

The UIDGenerator class is used to generate unique IDs for Ticket objects. It doesn’t quite fit the form of class declarations you’ve encountered so far. We won’t delve into the details; it suffices to show you a code fragment that demonstrates use:

```
//…
String UID_One = UIDGenerator.makeUID();
String UID_Two = UIDGenerator.makeUID();
System.out.println(UID_One);
System.out.println(UID_Two);
//…
```

The output of the fragment will be something like the following:

```
80000002
80000003
```

The specification for the new Clock class is as follows:

CONTINUED

Class Clock

A Clock object is used to simulate time and the passing of time.
A Clock can be examined and the time advanced. (It does not advance time on its own, hence "simulate".)

Instance variables

Time currentTime;

Constructors

Clock(Time time)
    // Create a Clock set to the given time.

Methods

public void advance(Duration duration)
    // Advance the clock time by the given duration.

public Time examine()
    // Obtain the current time (as recorded by this clock).

A Clock is mutable in that it stores a time value, and that time can be advanced.

Here's a snippet of code to illustrate behaviour:

```
//…
Clock c = new Clock(new Time("13:00"));
Time t = c.examine();
System.out.println(t.toString());
c.advance(new Duration("minute", 75));
System.out.println(c.examine().toString());
//…
```

The output from the fragment would be:

```
13:00:00
14:15:00
```

Finally, here is some more sample I/O for the CarParkSim class. Note the possibility that, when the user enters the 'depart' command, the given ticket ID might be invalid:

```
Car Park Simulator
The current time is 00:00:00.
Commands: advance {minutes}, arrive, depart, quit.
>advance 10
The current time is 00:10:00.
>arrive
Ticket issued: Ticket[id=80000001, time=00:10:00].
>advance 1
The current time is 00:11:00.
>arrive
Ticket issued: Ticket[id=80000002, time=00:11:00].
>arrive
Ticket issued: Ticket[id=80000003, time=00:11:00].
>depart 80000003
```

CONTINUED

```
Ticket details: Ticket[id=80000003, time=00:11:00].
Current time: 00:11:00.
Duration of stay: 0 minutes.
>depart 8000006
Invalid ticket ID.
>depart 80000001
Ticket details: Ticket[id=80000001, time=00:10:00].
Current time: 00:11:00.
Duration of stay: 1 minute.
>quit
Goodbye.
```

Exercise 2 [25 marks]

Your next task is to develop a TimePeriod class of object that meets the following specification:

Class TimePeriod

A TimePeriod is a Duration range. It has an inclusive lower bound, l, and an exclusive upper bound, u. A Duration, d, falls within the range if l<=d<u.

Constructors

public TimePeriod(Duration lowerBound, Duration upperBound)
// Create a TimePeriod with the given inclusive lower bound and exclusive upper bound.

Methods

public Duration lowerBound()
// Obtain the lower bound for this time period.

public Duration upperBound()
// Obtain the upper bound for this time period.

public boolean includes(Duration duration)
// Determine whether the given duration falls within this time period i.e. whether
// lowerBound()≤duration<upperBound().

public boolean precedes(TimePeriod other)
// Determine whether this time period precedes the other time period i.e. whether
// this.upperBound()≤other.lowerBound().

public boolean adjacent(TimePeriod other)
// Determine whether this time period is adjacent to the other time period i.e. whether
// this.upperBound() is equal to other.lowerBound(), or this.lowerBound() is equal to other.upperBound()

public String toString()
// Obtain a String representation of this TimePeriod object in the form:
// "[<duration> .. <duration>]".

If you look at the details for the `precedes()` and `adjacent()` methods you'll see that this is a 'self-referential' class declaration. Each of these methods, when performed on a TimePeriod object, accepts a reference to another TimePeriod objects as a parameter. (The Money, Time and Duration classes that you've been using for past assignments also have this characteristic – you're welcome to examine the code to see exactly how they function.)

CONTINUED

Here's a code snippet to illustrate TimePeriod behaviour:

```
//...
final TimePeriod pOne = new TimePeriod(new Duration("hour", 1), new
Duration("hour", 2));
final TimePeriod pTwo = new TimePeriod(new Duration("hour", 2), new
Duration("hour", 3));
final TimePeriod pThree = new TimePeriod(new Duration("hour", 3),
new Duration("hour", 4));
System.out.printf("%s\n%s\n%s\n",pOne, pTwo, pThree);

System.out.println(pOne.includes(new Duration("minutes", 59)));
System.out.println(pOne.includes(new Duration("minutes", 60)));
System.out.println(pOne.includes(new Duration("minutes", 119)));
System.out.println(pOne.includes(new Duration("minutes", 120)));

System.out.println(pOne.precedes(pThree));
System.out.println(pTwo.precedes(pOne));

System.out.println(pTwo.adjacent(pOne));
System.out.println(pOne.adjacent(pThree));
//...
```

Here's the output:

```
[1 hour .. 2 hours]
[2 hours .. 3 hours]
[3 hours .. 4 hours]
false
true
true
false
true
false
true
false
```

NOTE: To implement the `toString()` method you should use the Duration class format method. The specification is in the appendices.

How are you to evaluate your work? There are three possibilities: (i) write a test program, (ii) write Junit tests, or (iii) use the jGrasp interactive feature.

Exercise 3 [20 marks]

Your next task is to develop a ParkingTariff class. The primary objective is actually to develop a TariffTable, however, there's a design issue that must be solved first.

A TariffTable must store a series of parking tariffs. Each is an association between a time period and a cost. You could implement a TariffTable by using two arrays, one containing TimePeriod objects and the other containing Money objects, where the TimePeriod at index i in the first array is associated with the Money object at index I in the second array.

That's not a particularly nice solution since there's nothing in the resulting variable declarations to indicate the relationship, and there's potential for mismatch. A better solution is to have a TariffTable contain a collection of ParkingTariff objects, where each ParkingTariff object stores a TimePeriod and Money.

Develop a ParkingTariff class of object.

- Your class will have instance variable(s), constructor(s), and method(s).
- You should aim to move beyond simple get and set methods to ones that offer greater functionality.

With respect to the second point, you may wish to develop the ParkingTariff class in conjunction with the TariffTable class (exercise 3):

- Think about what a TariffTable object must do.
- Think about what could be delegated to ParkingTariff objects

Evaluate your work by (i) writing a test program, or (ii) writing Junit tests, or (iii) using the jGrasp interactive feature.

Your solution will be manually marked by the tutors.

Exercise 4 [30 marks]

Develop a TariffTable class that meets the following specification:

Class TariffTable

A TariffTable records parking tariffs for a pay-to-stay car park.

Constructors

public TariffTable(int maxSize)

// Create a TariffTable with the given maximum number of entries.

Methods

public void addTariff(TimePeriod period, Money tariff)

// Add the tariff for the given period to the table. The period must directly follow, and be
// adjacent to, that for the previous tariff entered.
// If the period does not follow or is not adjacent then an IllegalArgumentException is thrown.

public Money getTariff(Duration lengthOfStay)

// Obtain the tariff for the given length of stay.

public String toString()

// Obtain a String representation of this TariffTable in the form:
// <period$_0$> : <tariff$_0$>
// …
// <period$_n$> : <tariff$_n$>

Here's a snippet of code to illustrate behaviour:

```
//...
final Currency currency = new Currency("R", "ZAR", 100);
final TimePeriod pOne = new TimePeriod(new Duration("hour", 1), new
Duration("hour", 2));
```

CONTINUED

```
final TimePeriod pTwo = new TimePeriod(new Duration("hour", 2), new
Duration("hour", 3));

final TariffTable tariffTable = new TariffTable(2);
tariffTable.addTariff(pOne, new Money("R2", currency));
tariffTable.addTariff(pTwo, new Money("R5", currency));

System.out.println(tariffTable);

System.out.println(tariffTable.getTariff(new Duration("minute",
65)));
System.out.println(tariffTable.getTariff(new Duration("hour", 2)));
//...
```

The output from the fragment would be:

```
[60..120 minutes] : R2.00
[120..180 minutes] : R5.00
R2.00
R5.00
```

NOTE:

- The `toString()` method must return a string containing a series of lines, one for each parking tariff. All but the last line must end with a newline character '\n'.
- The addTariff() method must check that the period, p_n, for the new tariff follows on from the previous one, p_p entered i.e. that p_0 precedes and is adjacent to p_n. If this condition is not met then an exception should be thrown. Here' s the expression your code should use:

  ```
  throw new IllegalArgumentExeption("TimePeriod:addTariff():
  precondition not met.");
  ```

Evaluate your work by (i) writing a test program, or (ii) writing Junit tests, or (iii) using the jGrasp interactive feature.

Submission

Submit the `TimePeriod.java, ParkingTariff.java, TariffTable.java` and completed `CarParkSim.java` files to the automatic marker.

The ParkingTariff class will be assessed by the tutors. They will look at the class design and implementation and at the use of ParkingTariff objects in the TariffTable class.

CONTINUED

ParkingTariff		
Component	*Marks*	*Comment*
Fields	4	Uses appropriate types of fields with encapsulation.
Constructor	2	Uses suitable types of parameter.
Methods	6	Rich methods, e.g. a toString(), that support the uses of ParkingTariffs in the simulation - printing tariff details, and calculating cost of stay
TariffTable		
Field(s)	4	Uses a collection of ParkingTariff objects.
Methods	4	Stores, searches and, retrieves, and manipulates ParkingTariff objects.

By “most appropriate” and “suitable” we mean you should be thinking in terms of rich concepts such as Tickets, Time, TimePeriod, obtain cost of stay, printing tariffs etc, rather than Strings and ints etc i.e. ‘OO’ thinking.

Appendices

Class Money

An object of this class represents an amount of money in a particular currency. Amounts can be added and subtracted. The amount is stored as a quantity of the minor unit of the currency e.g. 1 Rand will be stored as 100 cents.

Constructors

public Money(String amount, Currency currency)

// Create a Money object that represents the given amount of the given currency.
// The String is assumed to have the following format: <currency symbol><quantity of
// units>.<quantity of minor units> e.g. in the case of USD, $50.30, $0.34.

Methods

public Money add(Money other)

// Add the other amount of money to this amount and return the result. The objects must be of the
// same currency.

public String toString()

// Obtain a string representation of the monetary value represented by this object e.g. "€45.10" for
// a Money object that represents 45 euros and 10 cents.

Class Currency

An object of this class represents a Currency such as US Dollars or British Pound Stirling.

A currency has an ISO 4217 currency code and a symbol denoting the currency's major unit. Many currencies have a minor (or fractional) unit such as the cent in the case of US dollars.

It is assumed that the currency symbol always appears in front of an amount; that negative amounts are represented by a minus sign, '-', that precedes the currency symbol, "-£34.50" for example; that the decimal point is always represented using a full stop; that no attempt is made to group the digits of large quantities, so for example, one million Rand is assumed to be represented as "R1000000" (as opposed to "R1,000,000").

Constructors

public Currency(String symbol, String code, int minorPerMajor)

// Create a Currency object that represents the currency with the given unit symbol (e.g. "£" for
// Sterling), ISO 4217 code, and number of minor units per major units (e.g. 100 in the case of
// pennies per British Pound).

CONTINUED

Class Time

A Time object represents a twenty-four-hour clock reading composed of hours, minutes and seconds.

Constructors

Time(String reading)

// Create a Time object from a string representation of a twenty-four-hour clock reading
// of the form 'hh:mm[:ss]' e.g. "03:25", "17:55:05".

Methods

public Duration subtract(Time other)

// Set the first, middle and last names of this Student object.

public String toString()

// Obtain a String representation of this Time object in the form "HH:MM:SS".

Class Duration

A Duration object represents a length of time (with millisecond accuracy).

Constructors

public Duration(String timeUnit, long quantity)

// Create a Duration object that represents the given quantity of the given time unit.
// Permissible time units are: "millisecond", "second", "minute", "hour", "day" ,"week".

Methods

public int compareTo(Duration other)

// Returns a negative, zero, or positive value, depending on whether this duration is smaller, equal
// to, or greater than the other duration.

public boolean equals(Object o)

// Determine whether object o is equivalent to this object i.e. if it is a Duration and of the same
// value as this Duration.

public static String format(final Duration duration, final String smallestUnit)

// Obtain a formatted string that expresses the given duration as a series of no-zero quantities of
// the given time units.
// For example, given a Duration, d, representing 88893 seconds, the expression
// Duration(d, "hour", "minute", "second") returns the string "24 hours 41 minutes 33 seconds",
// while Duration(d, "hour", "second") returns the string "24 hours 2493 seconds".