CSC1016S Assignment 5

Simple Classes with Constructors, Fields, and Methods

Introduction

This assignment concerns reinforcing OO concepts through creating and manipulating types of object; and the writing of simple class declarations.

Key points are that an OO programmer (i) often uses predefined program components i.e. classes, (ii) often develops program components, not whole programs and (iii) needs techniques and tools for checking/evaluating their work.

This assignment is a continuation of *Assignment 4*. Exercises one, two and three involve constructing class declarations for simple types of object. Exercise one uses the classes provided for exercise one, exercise two uses the class developed in exercise one, and exercise three uses the class developed in exercise two.

Furthermore**,** in this assignment, your solutions will be evaluated for correctness and for the following qualities:

- The use of object types, object creation, and the reading and writing of object fields.
- Documentation
    - Use of comments at the top of your code to identify program purpose, author and date.
    - Use of comments within your code to explain each non-obvious functional unit of code.
- General style/readability
    - The use of meaningful names for variables and functions.
- Algorithmic qualities
    - Efficiency, simplicity

These criteria will be manually assessed by tutors and commented upon. Up to 10 marks will be deducted for deficiencies.

The Scenario

The exercises are themed. They concern modelling aspects of a pay-to-stay car. 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.

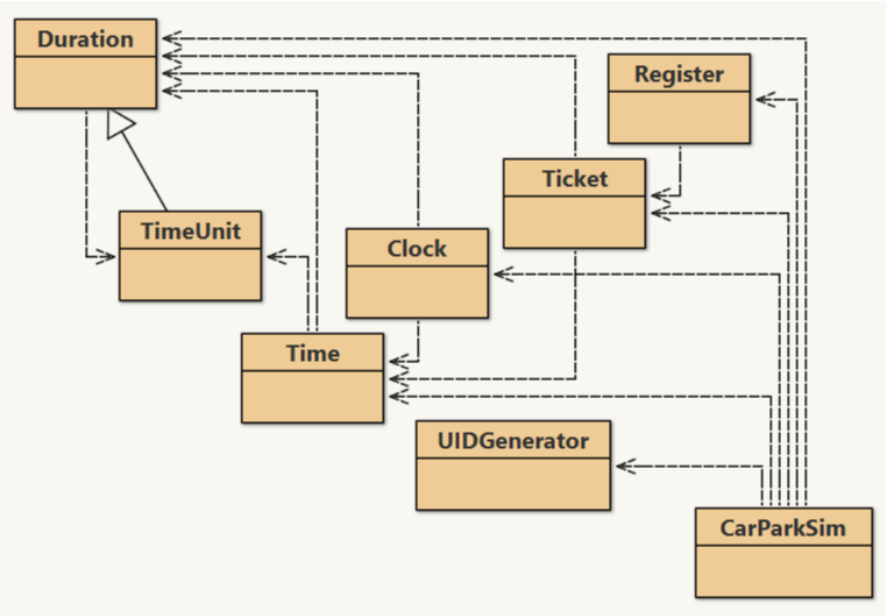
The final exercise completes the program. Here's a sample of program I/O:

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

Here is the design:



On the assignment page you will find Time, Duration, Clock, TimeUnit and UIDGenerator classes, and a skeleton implementation of the CarParkSimulator class – your task for exercise 3 is to fill in the blanks.

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.

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 Amathuba 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 Amathuba page for this assignment.

Exercise One [30 marks]

Your task is to develop a `Ticket class` of object. A Ticket object represents a car park ticket. It has a unique ID and time of issue (24-hour clock).

Your class must meet the following specification:

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

Instance variables
String id;
Time issueTime;

Constructors
Ticket(Time currentTime, String ID)
// Create a new Ticket that has the given issue time and unique ID.
Methods
public String ID()
// Obtain this Ticket's ID.
public Duration age(Time currentTime)
// Obtain this ticket's age i.e. the issue time subtracted from the given time.
public String toString()
// Obtain a String representation of this Ticket object in the form:
// "Ticket[id="ddddd", time="hh:mm:ss"]".

Here's a code snippet to illustrate behaviour:

```
//..
Time tOne = new Time("6:50");
Ticket ticket = new Ticket(tOne, "8005A3");
Time  tTwo = new Time("7:19");
System.out.println(ticket.toString());
Duration d = ticket.age(tTwo);
System.out.println(d.intValue("minute"));
//…
```

The output from this code would be:

```
Ticket[id=8005A3, time=06:50:00].
29
```

The class has a single constructor that is used to (i) assign the current time as the ticket issue time and to (ii) assign a unique ID.

Since this class is not in itself a complete program, how are you to evaluate your work? Consider (i) writing a test program (something like that required for *exercise 1 of Assignment 4*), or (ii) developing unit tests using Junit, or (iii) using the jGrasp interactive feature (*see Assignment 4)*.

Exercise Two [30 marks]

The job of a Register object in the car park 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 and calculate the duration of stay. Develop a `Register class` of an object based on the specification below:

Class Register
A Register stores a collection of Tickets. A Ticket may be retrieved given its ID.

Instance variables

Ticket[] tickets;
int numTickets;

Constructors

Register()
    // Create a new Register object.

Methods

public void add(Ticket ticket)
    // Store the given ticket in the register.
public boolean contains(String ticketID)
    // Determine whether a ticket with the given ID is in the collection.
public Ticket retrieve(String ticketID)
    // Get the Ticket with the given ID from the collection.

The idea is that, inside a Register object, there is an array in which Ticket objects are stored. Hence the instance variable "`tickets`". The variable "`numTickets`" stores the index of the next free space in the array.

- When a Register object is created, an array of Ticket object is created and assigned to `tickets`, and `numTickets` is set to zero.
- When a Ticket is added, it is put in the next free space (the value of `numTickets`), and `numTickets` is incremented.
- When a ticket is retrieved, the ticket ID is used to find the relevant Ticket object in the array and return it i.e. "`return tickets[i]`" where `i` is the relevant index.

The array should be of length 100.

Here is a snippet of code to illustrate behaviour:

```
//…
Register r = new Register();
Ticket t = new Ticket(new Time("13:00"), "00001");
String ID_One = t.ID();
r.add(t);
t = new Ticket(new Time("13:18"), "00002");
String ID_Two = t.ID();
r.add(t);
System.out.println(r.contains(ID_One));
System.out.println(r.contains("9236743"));
System.out.println(r.retrieve(ID_Two).toString());
//…
```

The output from the fragment would be:

```
True
False
Ticket[id=00002, time=13:18:00]
```

You can evaluate your work: by (i) writing a test program (something like that required for *Exercise 1, Assignment 4*), or (ii) by developing unit tests using Junit, or (iii) by using the jGrasp interactive feature.

NOTE: Clearly, using an array of length 100 as described is not the best solution for this class. When `numTickets==tickets.length`, the array is full. It's a simplification to make the exercise more accessible at this stage of the course. If you are confident of your Java skills you can try (a) a method for explicitly removing a Ticket from the array by inserting "`null`", and/or (b) dealing with the situation where the array fills up by making a bigger one, copying the contents of the old into it, and assigning it to `tickets`.

Exercise Three [40 marks]

On the assignment page you will find a skeleton implementation of the `CarParkSimulator class`. Your task is to fill in the blanks. Along with all the classes you've used and developed so far, you'll need the UIDGenerator class and the Clock class.

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:

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 (the input from the user is shown in **bold**). 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
```

```
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.
```

You may find the following slightly fuller specification for the Duration class useful.

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 non-zero time unit
// quantities from the largest applicable to the smallest.
// For example, given a Duration, d, representing 88893 seconds, the expression
// format(d, "second") returns the string "1 day 41 minutes 33 seconds",
// while format(d, "minute") returns the string "1 day 41 minutes".

Submission

Submit the `Ticket.java`, `Register.java`, and `CarParkSim.java` source files to the automatic marker in a ZIP file bearing your student number.

Appendices

Java Arrays

Arrays in Java bear a lot of similarity to Python Lists. Given an array, `A`, and an index value, `i`, a value, `v`, can be stored with the expression "`A[i]=v`". Similarly, a value can be retrieved, with the expression "`A[i]`" e.g. retrieving *v* and storing in a variable *n* is written "`n=A[i]`".

The differences are:

- An array stores a TYPE of value, which must be given when declared/described.
- An array is a type of object, and as such, is created using 'new'.
- An array has a fixed size which must be given when created.

Assume the following BMI class:

```
public class BMI {
	double height;
	int weight;

	double calculateBMI() {
		return weight/(height * height);
	}
}
```

Let's say we want an array that holds ten BMI objects. The following code snippet (i) declares a variable that can store an array of BMI, then (ii) creates such an array and assigns it to the variable:

```
//...
BMI[] records;
records = new BMI[10];
// ...
```

The variable declaration looks similar to others that we've used. The type is "`BMI[]`". It's the brackets that indicate the variable can store an array that stores BMI objects. Without the brackets, of course, it would just be a variable that can store a BMI object.

The creation expression is similar. The type of thing being created, "`BMI[10]`", is an array that can store BMI objects, the size of the array is ten.

Initially the array does contain any BMI objects. (The value at each index is the special value '`null`'.) Extending the code snippet as follows, we create a BMI object and insert it at location zero:

```
//...
BMI[] records;
records = new BMI[10];

BMI bmi_record = new BMI();
bmi_record.height = 165;
bmi_record.weight = 57;
records[0] = bmi_record;

System.out.println(records[0].height);
System.out.println(records[0].weight);
System.out.println(records[0].calculateBMI());
//...
```

The snippet ends with print statements. Each accesses the BMI object at location zero, i.e. this is what the expression "`records[0]`" does, and then one of the object's components. The first print accesses the `height` field, the second the `weight` field, and the third the `calculateBMI()` method.

For completeness, consider the following additional statements:

```
//...
System.out.println(records[0]);
System.out.println(records[1]);
//...
```

You might be inclined to think that the first statement prints out the BMI object stored at location zero, i.e. the height and weight. In fact, it prints something like the following:

```
BMI@7e6c04
```

The output consists of the name of the type of object (BMI) followed by an '@' sign, followed by what's called a "hashcode", a kind of identity code, and which we won't get into here. (If we had another BMI object and tried to print that we would generally get a different hashcode for it.)

The second print statement will output the following, since we haven't stored a BMI object at that location:

```
null
```

Finally, given an array, `A`, we can obtain its length with the expression "`A.length`".