Tutorial 3: Object-oriented concepts

1. Tutorial Objectives

In the following tutorial, we will learn about classes, objects and methods. Given the importance of object-oriented concepts, this tutorial moves smoothly from a concept to another. Take the time to understand the details.

There are three levels of difficulty of the tutorial exercises marked with asterisks (*) as follows:

* Easy ** Medium *** Hard

It is important to do the tutorial exercises in order as they often rely on a previous question to be completed first. Hints will be marked in italics.

You are expected to be able to finish the easy and medium exercises without any help. However, never hesitate to ask the demonstrator if you have any questions.

2. Exercises

This week's exercises provide you with an opportunity to define your own classes. As a reminder, here is a very simple class definition:

```
public class Circle {

	private double radius; // instance variable

	public Circle(double radius){   // constructor
		this.radius = radius;
	}

	public double getArea(){  // instance method
		return radius * radius * Math.PI;
	}
}
```

Recall that after the initial declaration, you should first define your instance variables, then write your constructor and finally define your instance methods.

* Exercise 1 Moving robots

We'll start by building a class **Robot1** for our first version of Robots.

We will think of a Robot as holding some data about itself and as also having certain behaviours. The data is captured by *instance variables* and the behaviours are captured by *instance methods*. In our first example, the only data held by a Robot is a battery charge, which we will represent as a *double*.

Robots will initially have two behaviours: they can get their batteries recharged (by a certain amount), and they can move a certain distance, based on the charge level of their batteries.

Figure 1: source http://www.telovation.com/photos/balancing-robot.jpg

Inside a **main**() method we can write a so-called *client code* to create new Robots and get them to do things. So, for example, here is how we might want to call some Robot methods, based on what we have so far.

```
public static void main(String[] args) {

 Robot1 r = new Robot1(); // start off with a well-charged battery
 r.move(11);             // move around and use up the charge
 r.batteryReCharge(2.5);  // get a new charge
 r.batteryReCharge(0.5);  // add a bit more
 r.move(5);              // move some more
}
```

Note: Please note that you need to create a launcher class ‘RobotTester’ that will contain your ***main()*** method. Also, you might want to organise your code in packages.

Once we have a reasonable idea of how the client code should work, we can start building the class that is called by the client. Here is a skeleton:

```
public class Robot1 {

	private double batteryCharge = 5.0; // instance variable

	public void batteryReCharge(double c) {
		// ADD CODE HERE
		}

	public void move(int distance) {
		// ADD CODE HERE
		}
}
```

Note that we've made the arbitrary decision that when a Robot rolls off the assembly line, its battery has a charge level of ***5.0***.

The next thing is to write the bodies of the instance methods. Implement **batteryReCharge()** so that the value of the argument is *added* to the existing *value of batteryCharge*. Remember that since *batteryCharge* is an instance variable of the class, you can use it directly inside any instance method. Next, implement the method **move()** so that it satisfies the following conditions:

- A Robot can only move when it has a battery charge greater than or equal to 0.5.
- For every unit of movement, the battery charge goes down by 0.5 units.

To make the exercise more interesting, get **batteryReCharge()** to print out the result of charging the battery. In addition, get **move()** to print out something to the terminal for every unit of movement that is executed, and to report if a flat battery is detected. Here's one way of doing it (based on the client code used earlier). The numbers in square brackets are meant to represent units of movement. However, you can do this in whatever way you like.

```
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] Out of power!
Battery charge is: 2.5
Battery charge is: 3.0
[1] [2] [3] [4] [5]
```

What have we learned from this very simple example of OOP? The main thing to take away is that data and behaviour can be closely interlinked. In this example, the effect of the method **move()** for Robot *d* depends on the state of *d* when **move()** is called; the number of units moved is dependent on the initial battery charge level. At the same time, behaviour changes the state of the object, since calling **move()** also depletes the battery charge level.

* Exercise 2 Talking robots

In our next robot example, we’re going to get the critters to talk. So let’s think about the class **Robot2**. Objects in this class will hold data about possible sayings, and have a method for speaking them; or more precisely, printing them out to the console! As before, we’ll start by considering what our client code might look like. Here’s an example:

```
public static void main(String[] args) {

	Robot2 r1 = new Robot2();
	String[] u1 = { "Exterminate, Exterminate!", "I obey!",
		"You cannot escape.", "Robots do not feel fear.",
		"The Robots must survive!" };

	r1.setSayings(u1);
	System.out.println("\nRobot r1 says: ");

	for (int i = 0; i < 10; i++) {
		r1.speak();
	}

	System.out.println("\nRobot r2 says: ");

	Robot2 r2 = new Robot2();
	String[] u2 = { "I obey!" };

	r2.setSayings(u2);

	for (int i = 0; i < 10; i++) {
		r2.speak();
	}
}
```

So here we have created two Robots, r1 and r2, and assigned each of them their own list of sayings. While r1 has a good repertoire, r2 is obsessively obedient. So different Robots can hold different data, in the shape of an array of strings.

Given this client code, see if you can figure out how to implement **Robot2**, taking **Robot1** as a starting point (see ‘inheritance’ in http://www.tutorialspoint.com/java/java_inheritance.htm). Remember that instance variables receive a default value, which is *null* in the case of all reference types, including arrays. You may find it convenient to assign your *sayings* the empty array as initial value.

Define **speak()** so that each time it is called, it picks one of the available sayings at random.

Hint: Use the Random class discussed in the lecture.

** Exercise 3 Credit Card

In this question, you are going to build a class representing a credit card. The exercise will give you more practice in manipulating strings. It also requires you to use the Java Calendar API in order to get the current year and month, which in turn will tell you whether a card has expired or not.

In a more realistic implementation, we would do a lot more checking of the validity of things like dates and credit card numbers! However, we won't bother with such details right now.

The **CreditCard** data type will contain the following data:

- an *expiryMonth* and an *expiryYear*, both of type *int*; these will determine the card's expiry date.
- a *firstName* and a *lastName*, both of type String; these will determine the name of the card's account holder.
- and finally, a *ccNumber*, the *String* that holds the card's number.

You should implement all these as private instance variables.

Here is the API for the class:

public CreditCard(int expiryMonth, int expiryYear, String firstName, String lastName, String ccNumber)

The class constructor; this will set the values for the class's instance variables. Here's an example of initializing a new instance of this class:

```
CreditCard cc1 = new CreditCard(10, 2014, "Bob", "Jones",
"1234567890123456");
```

Notice that we are passing the *expiryYear* argument as a four-digit number, and that *ccNumber* is a 16-character string.

public String **formatExpiryDate()**: Return the expiry date (month and year) in the format 'MM/YY'. For example, calling **formatExpiryDate()** on *cc1* should return the string "10/14". The month does not need to be zero-padded.

public String **formatFullName()**: Return the full name of the account holder in the format '*firstName lastName*'. For example, calling **formatFullName()** on *cc1* should return the string "Bob Jones".

public String **formatCCNumber()**: Return the credit card number as a four-block string, where blocks are separated by a single whitespace. For example, calling *formatCCNumber()* on *cc1* should return the string "1234 5678 9012 3456".

public boolean **isValid()**: Returns *true* if the expiry date of the credit card is later than the current value provided by the Calendar utility. For example, calling **isValid()** on *cc1* would return *true* if evaluated at date October 2015 or any prior date.

public String **toString()**: Returns a multi-line *String* which contains information such as the credit card number, the account holder, the expiration date and whether the card is valid or not. Here's an example of the expected output:

```
Number: 1234 5678 9012 3456 Expiry date: 10/14 Account holder:
Bob Jones Is valid: true
```

In order to determine the current date, you will need to include the following code in your class definition:

```
import java.util.Calendar;

Calendar now = Calendar.getInstance();
```

In order to get the current year, call *now.get(Calendar.YEAR)*; and similarly for the current month.

When you write the code for ***isValid()***, do not use the ***Calendar.after()*** method; instead, write your own boolean test.

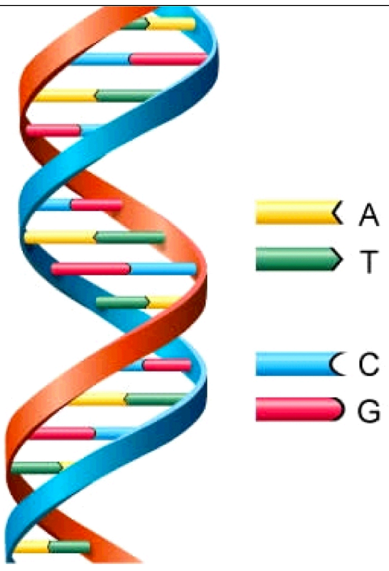
Write a **CreditCard** class which meets the API above.

In addition, we recommend that you write a client program **CreditCardTester**. This should contain a **main()** method which creates some **CreditCard** objects (one valid and one invalid) and calls the methods of these objects.

** Exercise 4 DNA

Our genetic makeup, and that of plants and animals, is stored in DNA. DNA is composed of two DNA-Strands, which twist together to form a double helix.

Each strand of DNA is a chain of smaller bases connected in a line. There are four bases:

• adenine (abbreviated A) • thymine (T) • cytosine (C) • guanine (G)	 Figure 2: Source http://spittalstreet.com/?p=4229

The bases have pairs: A binds to T and C binds to G. The two strands ‘match up’, so that the bases at each point bind to their corresponding pairs. So for example, if one strand was

A-C-G-G-T-C

then the other strand would be:

T-G-C-C-A-G

Since we have the pairs A:T, C:G, G:C, G:C, T:A and C:G. This may sound like a redundant way of storing information, but it means that if we pull apart the strands of the double helix and throw one side away, we are still able to reconstruct it. Furthermore, if we want to duplicate our double helix, we can split it into the two strands, reconstruct each side, and then end up with two copies of the original double helix.

In this exercise, we will create a class that represents a strand of DNA.

Here is the API for the class **DNAStrand**:

public **DNAStrand(String dna)**: Class constructor.

public boolean **isValidDNA()**: Returns true if the DNA is valid, i.e, only contains the letters, A,T,C,G (in uppercase) and at least one of these characters.

public String **complementWC()**: Returns the Watson Crick complement, which is a string representing the complementary DNA strand (i.e., the other strand in the double helix). So swap all T's with A's, all A's with T's, all C's with G's and all G's with C's.

public String **palindromeWC()**: Returns the Watson Crick Palindrome, which is the reversed sequence of the complement.

public boolean **containsSequence(String seq)**: Returns true if the DNA contains the subsequence seq.

public String **toString()**: Returns the underlying DNA sequence string.

Write a **DNAStrand** class which meets this API.

In addition, create a client class **DNAStrandTester** which contains the following static method:

```
public static void summarise(DNAStrand dna) {

      System.out.println("Original DNA Sequence: " + dna);

      if (dna.isValidDNA()) {
            System.out.println("Is valid");
            System.out.println("Complement: " + dna.complementWC());
            System.out.println("WC Palindrome: " +
                                    dna.palindromeWC());
      }  else {
            System.out.println("Not Valid DNA");
      }
}
```

Create one or more **DNAStrand** objects inside the **main()** method of **DNAStrandTester** and test them with the **summarise()** method.

~~~ ooo ~~~
~~~