CSC1016S Assignment 2

Class declarations – Creating a Basic Class

Assignment Instructions

This assignment involves building, testing and debug Java programs, that create and manipulate composition of user-defined types of object comprising public instance variables ONLY.

The goal is to form class declarations and use the dot notation to access and use their instance variables.

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 20 marks will be deducted for deficiencies.

Exercise 1 [40 Marks]

Write a class called "Seller" that may be used to model shops with the following attributes:

Attribute name	**Stored value**	**Description**
ID	PNP01	A unique key of the seller
Name	Pic n pay	Name of the seller
Location	Rondebosch	Location of the seller
Product	Meet	The product to sell
unit_price	R49.99	Price per unit
number_of_units	1000	Number of available units

The class will only contain instance variables (we'll move on to constructors and methods in later assignments), one per attribute. Each variable should bear exactly the same name as the attribute it models. You must choose a suitable type for each variable based on the sample values. Hint: Money and Currency from assignment 1?

Based on the following sample I/O, write a simple program called "`TestSeller.java`" that asks the user to input seller details (as specified in the table), creates a Seller object and inserts the details in the appropriate fields, then retrieves and prints out those details.

Sample IO *(The input from the user is shown in* ***bold*** *font – do not program this):*

```
Please enter the details of the seller.
ID: UCT01
Name: UCT
Location: Rondebosch
Product: UCT T-shirts
Price: R300.00
Units: 120
The seller has been successfully created:
ID of the seller: UCT01
Name of the seller: UCT
Location of the seller: Rondebosch
The product to sell: UCT T-shirts
Product unit price: R300.00
The number of available units: 120
```

Exercise 2 [60 marks]

This exercise follows on from the previous. It concerns writing a program called "`MarketPlace.java`" that creates a collection of Seller objects from a data file, and then searching for and printing those that sell a product meeting the user's criteria for quantity and price.

The program will ask the user to enter the name of a "Comma Separated Values" (CSV) file containing seller data. The first line of the file will consist of an integer `N` that represents the number of sellers.

Each subsequent line will consist of the comma separated values of the attributes for a single seller.

<shop identifier>, <seller name>, <product>, <price per unit>, <number of units available>

For example,

```
8
KP01, Kwik Pic, Bolton Hill, Cauliflower, R10.00, 120
FM01, Farmers' Market, Imizamo Yethu, Lettuce, R11.25, 132
BAL01, Buy-A-lot, Salt River, Concrete Block, R3, 900
NA01, NotAllot, Claremont, Hot Air, R99.99, 0
STWD01, ShopTillWeDrop, Cape Town CBD, Lettuce, R10.00, 100
24S01, 24-Seven, Milnerton, Fish fillets, R21.50, 132
MM01, Mega Mart, Phillipi, Lettuce, R13.00, 132
GNG01, GetnGo, Rantanga Junction, Lettuce, R9.00, 50
```

The program will:

1. Read in and create an array of Seller objects from the data.
2. It will then ask the user to enter the name of a product, the quantity required, and the maximum that they are willing to pay per unit.
3. It will then search the array, printing out details of those sellers that can meet the requirements.

If there are no matches, then it will print "`None found.`".

You will need to use Java arrays and know how to use Scanner objects to process input from file. Explanations may be found in the appendices at the end of this document.

Sample IO *(The input from the user is shown in* ***bold*** *font – do not program this):*

```
Enter the name of a "Comma Separated Values" (CSV) file:
sellers.csv
Enter the name of a product:
Lettuce
Enter the number of units required:
87
Enter the maximum unit price:
R12.00
FM01 : Farmers's Market in Imizamo Yethu, has 132 Lettuce for R11.25 each.
STWD01 : ShopTillWeDrop in Cape Town CBD has 110 Lettuce for R10.00 each.
```

Sample IO *(The input from the user is shown in* ***bold*** *font – do not program this):*

```
Enter the name of a "Comma Separated Values" (CSV) file: mydata.csv
The file contains no seller data.
```

Sample IO *(The input from the user is shown in* ***bold*** *font – do not program this):*

```
Enter the name of a "Comma Separated Values" (CSV) file:
shops.csv
Enter the name of a product:
Fire lighters
Enter the number of units required:
10
Enter the maximum unit price:
R5.00
None found.
```

Submission

Submit the `Seller.java`, `TestSeller.java`, and `MarketPlace.java` files to the automatic marker.

NOTE: The assignment is partially assessed automatically. Tutors will manually assess your use of the Seller class [10 marks], and the reading and writing of object fields [10 marks] in MarketPlace.java.

Appendices

The following sections get you started with Java arrays, and processing input from file with Scanner objects.

Money Class

The Money class of object has a "`compareTo()`" method which can be used to determine if one Money object is greater, equal or less than another.

Methods

public int compareTo(Money other)

Compare this Money object to the other Money object, returning a negative, zero, or positive value depending on whether this Money object is smaller, equal to, or larger than the other Money object.</p>
The Money objects that are compared must be of the same currency.</p>

For example, given the following code snippet:

```
//...
Currency rand = new Currency("R", "ZAR", 100);
Money mOne = new Money("R25.70", rand);
Money mTwo = new Money("R50.50", rand);
Money mThree = new Money("R25.70", rand);
System.out.println(mOne.compareTo(mTwo));
System.out.println(mOne.compareTo(mThree));
System.out.println(mTwo.compareTo(mOne));
//...
```

The output will be:

```
-1
0
1
```

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`".

Scanner objects for file processing

The following skeleton shows how to create a Scanner that reads data from a file. It tests that there is at least one line, and if there is, reads and prints it:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class DemoProgram {
	public static void main(final String[] args)  throws FileNotFoundException
	{
		String fileName = "test.txt"
		// Get filename from user.
		Scanner fileIn = new Scanner(new File(fileName));
		if (fileIn.hasNextLine())
		{
			String firstLine = fileIn.nextLine();
			System.out.println(firstLine);
		}
		fileIn.close();
	}
}
```

The code can be generalised to read all lines by using a 'while-loop'. Note the import statements, the clause "`throws FileNotFoundException`" in the method signature, and that the Scanner must be closed after use.

A Scanner can also get its data from a String, and we can tell it how to split up the data into tokens. By default, a Scanner assumes tokens (items) in the stream of character data that it processes are separated by whitespace. There is a method for customising this.

The following code snippet creates a Scanner from a String, and tells it to split up the string according to the placement of colon, ":", characters.

Each call of "next...()" will return a bit of data delimited by colons.

```
\\...
Scanner scanner = new Scanner("Friday: 12:45");
scanner.useDelimiter("\\s*:\\s*");
System.out.println(scanner.next());
System.out.println(scanner.nextInt());
System.out.println(scanner.nextInt());
\\...
```

The snippet prints:

```
Friday
12
45
```

The second statement is where we tell the Scanner to use colon as a separator. Actually, we've done something a bit more complicated.

The code actually tells the Scanner that parts are separated by a sequence of zero or more spaces, followed by a colon followed by zero or more spaces. This makes the code a bit more robust.

Useful Scanner methods: `hasNextLine()`, `nextLine()`, `hasNext()`, `next()`, `hasNextInt()`, `nextInt()`.

END