COSC1284 Programming Techniques
Semester 2 (2019)
Assignment 1

Ref: http://pngimg.com/download/55768

Due:	**16th August, 2019 (Friday)** **i.e. end of week 4** **Late submissions accepted until** **21st August, 2019 with penalty** **11:59 pm AEST**
Assignment Type:	**Individual (Group work is not permitted)**
Total Marks:	**8 marks (8%)**

Background information

For this assignment you need to write an object-oriented console application in the Java programming language which adheres to the following object-oriented programming principles:

a) Follow good object-oriented principles such as encapsulation, composition & cohesion.

b) Set the visibility of all instance variables to private.

c) Use getter and setter methods only where needed and appropriate with consideration given to scope (visibility).

d) Only use static attributes and methods when there is a good reason to do so, such as reducing code repetition whilst still maintaining good object oriented design.

e) Encapsulate both the data for a class and the methods that work with and/or manipulate the data within the same class.

f) Use superclass methods to retrieve and/or manipulate superclass properties from within subclass methods. Avoid accessing variables directly.

Overview

For this assignment you need to write a console application in the Java programming language which simulates a food delivery service called **MiChef**.

The staff at **MiChef** will need to be able to log the details for deliveries that are available for **MiChef's** customers.

Assignment 1 – Getting Started

This assignment is divided up into a few stages, each of which tackles concepts discussed during the course, as shown below:

- Stage 1 - Customer class (Classes & String Processing) (2 marks)
- Stage 2 - Meal class (Arrays) (4 marks)
- Stage 3 - Delivery class (Composition) (2 marks)

The assignment is designed to increase in complexity with earlier stages being easy and the later stages becoming increasingly more difficult. In addition, the specification will be more prescriptive in the earlier stages. The later stages of the assignment are less prescriptive and will require you to be more independent and to resolve design issues on your own.

Note:

- You are not permitted to use streams from Java 1.8.
- Marking of the assignment will be done using Java 1.8.
- You must use an array to store any collections of data in your program.
- You must not use ArrayLists, Maps or any other data structures within the Java API or from 3rd party libraries.

Marking Penalties

- Failing to comply with the above restrictions will incur a 50% marking penalty.
- Code that is submitted with compilation errors will incur a 50% marking penalty.
- Code that produces a runtime error and causes the program to crash will incur a 25% marking penalty.

- Throughout this specification, if the signature of method is shown, then you must implement the method with the exact signature as shown.

Disclaimer:

While the scenario described is based upon the concept of a meal delivery service, the specification for this task is intended to represent a simplified version of such a system and it is not meant to be a verbatim simulation of any such meal delivery service.

Stage 1 - Customer (Design / Implementation) (2 marks)

Some specific instructions are given below to get you started with the basics, but you will need to use the knowledge and skills you are developing in the course to further develop these classes and solve any problems you encounter as required.

You need to make sure that any modifications/changes you make adhere to good object-oriented concepts and design.

A) Create the Customer Class

The **Customer** class represents a single customer.

The following attributes should be defined in your **Customer** class.

You should **NOT** define any additional instance variables unless strictly necessary.

Instance variable	Type
`firstName`	`String`
`lastName`	`String`
`streetNo`	`String`
`streetName`	`String`
`suburb`	`String`
`postcode`	`String`

B) Constructor

```
public Customer(String firstName, String lastName,
                String streetNumber, String streetName, String suburb, String postcode)
```

The constructor is responsible for initialising the object into a valid state at the time of construction.

You should **NOT** have a default constructor.
You should **NOT** modify the definition of this constructor.

Business Rules:

1. Attributes are not permitted to have an empty string assigned to its value nor can they be assigned null values.
2. The postcode must be exactly four characters, consists only of numeric characters and the first character must be a digit from 1 - 8.
3. If any of the above rules are violated, then the value of the relevant attribute should be set to "N/A"

C) Implement a method for getting all the details of a Customer object

public String getDetails()

This method should build and return a string. The returned string should be formatted in a human readable format as shown below. This method should not do the actual printing.

The description must include labels and values for all the instance variables as shown in the examples below. Note these are just a couple of sample outputs.

Sample output for a customer with **VALID** data

```
Name:        Henry Cavill
Address:     83 Dalgliesh Street
             South Yarra 3141

Name:        Matthew Broderick
Address:     42 Pride Avenue
             Elwood 3184
```

Sample output for a customer with **INVALID** data
(empty street name and a post code that contains letters, or the first character is a 0 or 9.)

```
Name:        Angelina Jolie
Address:     83 N/A
             South Yarra N/A
```

D) Implement a toString method

public String toString()
This method should override the Object.toString() method. It should build a string and return it to the calling method. The returned string should be formatted in a pre-defined format designed to be read and processed by a computer.

Format for the toString method is a colon separated list of the instance variables as shown below:

```
firstName:lastName:streetNo:streetName:suburb:postcode
```

Sample output for a customer with **VALID** data.

```
Henry:Cavill:83:Dalgliesh Street:South Yarra:3141

Matthew:Broderick:42:Pride Avenue:Elwood:3184
```

Sample output for a customer with **INVALID** data
(empty street name and a post code that contains letters, or the first character is a 0 or 9.)

```
Angelina:Jolie:83:N/A:South Yarra:N/A
```

E) Write a class to test your customer class

Write a class called **TestCustomer** that has two methods called **'testValid'** and **'testInvalid'**.

In these methods you should **FULLY** test all the business rules of the class. The test data in this class should be hard coded and not provided by user input.

In the 'testValid()' method:

- Create the following customers and write statements to print out the getDetails and the toString results for each customer:

```
Matthew:Broderick:42:Pride Avenue:Elwood:1184
Rowan:Atkinson:91:Sebastian Street:Carlton:2053
Jeremy:Irons:19:John Close:Essendon:3040
Whoopi:Goldberg:57:Elaine Court:St Albans:4021
James:Jones:11:Earl Road:Melbourne:5000
Henry:Cavill:83:Dalgliesh Street:South Yarra:6141
Angelina:Jolie:11:Smith Street:Toorak:7142
Matt:Bomer:42:Staton Street:South Yarra:8141
```

```
Note: Post codes do not match actual suburbs, but used for testing
purposes to validate business rules.
```

In the 'testInvalid()' method

Create the following customers and write statements to print out the getDetails() and the toString() results for each customer:

- Create a customer with all the attributes set to an empty string;
- Create a customer with a postcode that has alphabetical characters;
- Create a customer with a postcode that is less than 4 characters;
- Create a customer with a postcode that is more than 4 characters;
- Create a customer with a postcode that has a first character '0';
- Create a customer with a postcode that has a first character '9'.

In the **main** method of your program:

- Instantiate the TestCustomer class and call both methods.
- Examine the results of the printing to ensure your class is working correctly and displaying the correct output.

For example, the toString() results for the invalid data should be:

```
N/A:N/A:N/A:N/A:N/A:N/A
Rowan:Atkinson:91:Sebastian Street:Carlton:N/A
Jeremy:Irons:19:John Close:Essendon:N/A
Whoopi:Goldberg:57:Elaine Court:St Albans:N/A
James:Jones:11:Earl Road:Melbourne:N/A
Angelina:Jolie:83:Smith Street:Toorak:N/A
```

Backup your program once the testing is complete.

Stage 2 - Meal (Design / Implementation) (4 marks)

Make a backup of your program before continuing

YOU MUST COMPLETE STAGE 1 BEFORE ATTEMPTING STAGE 2

IF STAGE 1 HAS NOT BEEN COMPLETED, STAGE 2 WILL NOT BE ASSESSED

In this section of the assignment we will list your requirements, but it will be up to you to make decisions and to design an actual solution.

A) Create the Meal Class

The **Meal** class represents a single meal and has the following attributes:

Name, meal category, list of basic ingredients, list of additional ingredients, cost per single serve

B) Constructor

```
public Meal(String name, MealCategory category, String[] ingredients, double cost)
```

The constructor is responsible for initialising the object into a valid state at the time of construction.

You should **NOT** have a default constructor.
You should **NOT** modify the definition of this constructor.

Business Rules:

1. Attributes are not permitted to have an empty string values. Values will be changed to "N/A" if an empty string is passed into the constructor.
2. A meal must have a minimum of two ingredients.
3. If any of the ingredients are duplicated, the ingredients list is set to null.
4. If the ingredients list contains null values, the null values are removed.
5. Each array must only be large enough to accommodate the number of items it holds.
6. The minimum cost for a meal is $5.00 and an invalid value will be set to -1.00.
7. The meal category should be a fixed list of possible categories available: INDIAN, VIETNAMESE, CHINESE, KOREAN, JAPANESE, EUROPEAN, & MIDDLE EASTERN.

C) Implement a method for adding ingredients

```
public boolean addIngredient(String ingredient)
```

You should **NOT** modify the definition of this method.

Business Rules:

1. Added ingredients can only be added to the additional ingredient list and not the basic ingredient list.
2. There cannot be duplicate ingredients across either of the two lists.

The **add ingredient** method should add the ingredient to the list of additional ingredients and add 50 cents to the cost of the meal.

The addition of the ingredient should not be permitted if the ingredient already exists in either the basic ingredient list or the custom ingredient list.

D) Implement a method for adding ingredients

```
public boolean removeIngredient(String ingredient)
```

You should **NOT** modify the definition of this method.

The **remove ingredient** method should remove an ingredient from either ingredient list and only reduce the cost of the meal by 50 cents if the ingredient being removed is from the list of additional ingredients.

You must ensure that no business rules are violated as a result of these operations and return an appropriate value from the method depending on the success or failure of the operation.

E) Implement a method for getting all the details of a Meal object

```
public String getDetails()
```

You should **NOT** modify the definition of this method.

This method should build and return a string. The returned string should be formatted in a human readable format as shown below. This method should not do the actual printing.

The description must include labels and values for all the instance variables as shown in the examples below. Note these are just a couple of sample outputs.

Sample output for a Meal **WITHOUT** additions

```
Name:        Salmon Soba
Category:    MIDDLE_EASTERN
Ingredients: Atlantic Salmon, Singapore Noodles, Capsicum, Broccoli
Cost:        $23.50
```

Sample output for a Meal **WITH** additions

```
Name:        Shakshouka
Category:    MIDDLE_EASTERN
Ingredients: Eggs, Tomatoes, Onions, Preserved Lemon, Garlic
Additions:   Chilli
Cost:        $35.50
```

Sample output for a Meal **WITH** invalid data less than 2 ingredients and costs less than $5.00

```
Name:        Congee
Category:    CHINESE
Ingredients: N/A
Additions:   Chilli
Cost:        $-1.00
```

D) Implement a toString method

public String toString()
This method should build a string and return it to the calling method. The returned string should be formatted in a pre-defined format designed to be read and processed by a computer.

Format for the toString() method is a colon separated list of the instance variables as shown below:

```
name:category:ingredients:additions:cost
```

Sample output for a meal **WITHOUT** additions.

```
Salmon Soba:VIETNAMESE:Atlantic Salmon, Singapore Noodles, Capsicum,
Brocolli:None:23.50
```

Sample output for a meal **WITH** additions.

```
Shakshouka:MIDDLE_EASTERN:Eggs, Tomatoes, Onions, Preserved Lemon,
Garlic:Chilli:17.25
```

Sample output for a Meal **WITH** invalid data less than 2 ingredients, cost less that $5.00

```
Congee:CHINESE:N/A:Chilli:-1.00
```

E) Write a class to test your meal class

Write a class called TestMeal that has five methods. This class should have a main method so that it can be run independently:

- testPassInstantiation
- testFailInstantiation
- testFailAddIngredient
- testPassAddIngredient
- testFailRemoveIngredient

In these methods you should **FULLY** test all the business rules of the class.
Test for every scenario you can think of and becareful to always allow for 'null' values.
The test data in this class should be hard coded and not provided by user input.

In this section of the assignment you are expected to design your own data to make sure that you fully test this class.

Examine the results of the printing to ensure your class is working correctly and displaying the correct output.

Backup your program once the testing is complete.

Stage 3 - Delivery (Design / Implementation) (2 marks)

Make a backup of your program before continuing

YOU MUST COMPLETE STAGE 1 & 2 BEFORE ATTEMPTING STAGE 3

IF STAGE 1 & 2 HAVE NOT BEEN COMPLETED, STAGE 3 WILL NOT BE ASSESSED

At this stage of the assignment you must make all the design and implementation decisions.

You should think carefully about how to make use of all the object-oriented concepts you have learned so far in the course.

The basic requirements of this class are as follows:

A) Creation of a Delivery Class

```
public Delivery(Customer customer)
```

The constructor is responsible for initialising the object into a valid state at the time of construction.

You should **NOT** have a default constructor.
You should **NOT** modify the definition of this constructor.

This class should create an object that represents a delivery that consists of a single customer and one or more meal objects.

B) Implement a method for adding meals to a delivery object

```
public void addMeal(Meal meal)
```

Once the construction process has finished, you should be able to add meals to the delivery.

C) Implement a method for setting the date of a delivery object

```
public boolean setDeliveryDate(int day, int month, int year)
```

There must be a mechanism by which you can tell if the delivery object is ready to be delivered.

The delivery should have a mechanism by which to set the day of the week on which the meal will be delivered.

The delivery should also have the actual date on which the meal will be delivered.

For example: The meal will be delivered on Wednesday, 24th July 2019.

You must use the provided DateTime class for working with dates.

D) Implement a method for getting all the details of a delivery object

public String getDetails()

E) Implement a toString method

public String toString()

F) Write a test class to fully test the Delivery class

In addition to hard coded data, you should also have a method that prompts the user for inputs so the class can be tested dynamically.

Coding Style

Your program should demonstrate appropriate coding style, which includes:

- **Formatting**

  Indentation levels of 3 or 4 spaces used to indent or a single tab provided the tab space is not too large - you can set up your IDE/editor to automatically replace tabs with levels of 3 or 4 spaces.

  A new level of indentation added for each new class/method/ control structure used.

  Indentation should return to the previous level of at the end of a class/method/control structure (before the closing brace if one is being used). Going back to the previous level of indentation at the end of a class/method/control structure (before the closing brace if one is being used)

  Block braces should be aligned and positioned consistently in relation to the method/control structure they are opening/closing.

  Lines of code not exceeding 80 characters in length - lines which will exceed this limit are split into two or more segments where required (this is a guideline - it's ok to stray beyond this by a small amount occasionally, but try to avoid doing so by more than 5-6 characters or doing so on a consistent basis).

  Expressions are well spaced out and source is spaced out into logically related segments

- **Good Coding Conventions**

  Identifiers themselves should be meaningful without being overly explicit (long) – you should avoid using abbreviations in identifiers as much as possible (an exception to this rule is a “generic” loop counter used in a for-loop).

  Use of appropriate identifiers wherever possible to improve code readability.

  All identifiers should adhere to the naming conventions discussed in the course notes such as ‘camel case’ for variables, and all upper case for constants etc.

  Complex expressions are assigned to meaningful variable names prior to being used to enhance code readability and debugging.

Commenting

Note: the examples provided below do not reflect actual comments you should have in your assignment, but are used for demonstration purposes only.

Class Level commenting

Each file in your program should have a comment at the top listing your name, student number and a brief description of the contents of the file (generally stating the purpose of the class)

```
/**
* <h1>Menu</h1>
* <p><b>This class provides a user interface to allow the user * to use different
examples of algorithms for sorting and
* filtering data.</b></p>
* <p><b>References: https://www.baeldung.com/java</b></p>
*
* @author  Student Name (s1234567)
* @version 1.0
* @since   2019-05-16
*/
```

Method Level commenting

All methods should be commented using Java docs.

```
/**
   * <p>This method enables the the insertion of indentation
  * levels into a string.</p>
   * <p>The method builds a string with tab characters equal to
  * the value provided.</p>
   * @param level This provides number of indents required.
   * @return String This returns a string with the specified
  * number of tab characters.
   */
```

Complex methods should have an algorithm as a multi-line comment. You don't need to specify test cases for every algorithm, only if it is appropriate.

```
/**
   * <b>This demonstrates how to calculate the sum of all
  * numbers from zero up to a given value.</b>
   * <p>This method uses iteration to perform the calculation.
  * Negative numbers are not supported.</p>
   * <pre>
  *    ALGORITHM - Iterative sum
```

```
*    BEGIN
*          DEFINE the upper bound
*          DEFINE container for storing the result
*          IF the upper bound less than zero
*                EXIT
*          END IF
*          REPEAT upper bound number of times
*                ADD upper bound to the current result
*          END REPEAT
*    END
 * </pre>
 * @param upperBound This provides the upper bound of the
* numbers being processed.
 * @return int This sum of all the numbers between zero and
* the upper bound.
 */
```

Code block commenting

Methods that contain multiple logic or code blocks should either be refactored into smaller discrete methods or provide inline commenting to identify and explain each logic or code block.

```
/* calculate the total value of the room rental
 * prior to applying surcharge.*/
int standardRate = 200;
int numberOfNights = 4;
int totalBeforeSurcharge = standardRate * numberOfNights;
```

Commented out code

Any incomplete or non-functional code should be removed from your implementation prior to your final submission.

Code that is not being used in your final implementation will attract marking penalties.

Examples of bad commenting

An example of a bad comment and/or coding style:

```
// declare an int value for storing the basic nightly rate for a room
int standardRoomRate = 200;

// declare and assign an initial account balance.
double x = 500.0;
```

What to Submit?

You should export your entire eclipse project to a zip archive and submit the resulting zip file - do this from within eclipse while it is running, not by trying to copy or move files around in the eclipse workspace directly, as you may corrupt your entire workspace if you do something wrong.

All students are also advised to check the contents of their zip files by opening them and viewing the files contained within before submitting to make sure they have done it correctly and that the correct (latest) version of the source code file is present to avoid any unpleasant surprises later.

You should also download a copy of your submission from Canvas after submission, so that you can double check that you have submitted the correct version.

Technical issues that result in the loss of your work or submitting an incorrect version **WILL NOT** be considered a basis for extensions or re-marking.

We are obliged to accept each submission in the form it is sent to us in, so make sure you submit the correct final version of your program!

The name of both your Eclipse project and your zip archive should be your student number followed by an underscore, then append A1. For example:

Project Name:	s123456_A1
Zip Archive:	s123456_A1.zip

Your submission will be penalised heavily if you fail to meet any of the above guidelines.

Please note if you make multiple submissions, canvas will change the name of the file you submit. That is OK and you should not worry about the name change.

Please refer to the extensions document for information on valid extension requests.