University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

AUTUMN TRIMESTER EXAMINATIONS

ACADEMIC YEAR 2020/2021

COMP20250: Introduction to Java
COMP20300: Java Programming (Mixed Delivery)

Exam Part A

Assoc Prof. Chris Bleakley
Dr. Simon Caton

Time Allowed: 1.5 Hours

Answer either question 1 or question 2.

The ExamResourcesPartA.zip file has the following contents:

1. **TestsQuestionOne.java** this contains the unit test cases for all coding parts of Question 1. These are indicative of a D or better grade.
2. **TestsQuestionTwo.java** this contains the unit test cases for all coding parts of Question 2. These are indicative of a D or better grade.
3. **Main.java** this is needed in Question 2, part c. You should complete this file. You may add any methods you wish, and also may introduce additional classes if needed. However, DO NOT change the signature of the constructor, or four accessor methods. Grading unit tests will only call the 4 accessor methods to test your code.

Question 1

Answer parts (a) to (d) with solutions coded in Java, and answer part (e) via video recording and demonstration.

35 marks in total

In this question, you will make **one** interface, and **three** classes:

1. an interface called **Building** in part a
2. a class called **House** in part b
3. a class called **Bungalow** in part c
4. a class called **Estate** in part d

The unit tests for this question are located in **TestsQuestionOne.java**

(a) (2 marks)
Create an <u>interface</u> called **Building**, it should have the following methods:

1. **addRoom** which receives a **double** and an **int** as input in this order. The **double** represents the area of the room and an **int** which represents the floor the room is on.
2. **getNumWindows** which returns an **int**: the number of windows a **Building** has.
3. **getFloorSpace** which returns a **double**: this method will calculate the area of all rooms on a given floor, i.e. sum them. The method should receive an **int**: indicating which floor to calculate the area for.
4. **getTotalFloorSpace** which returns a **double**: this will represent the calculation of the sum of floor space for all rooms on all floors

The unit test case for this question is called **testInterface**.

(b) (5 marks)
Create a class called **House** which implements the methods of **Building**, and has a constructor which receives an **int**: the number of floors the **House** has.

By default, **Houses** have 2 windows per floor. Every time a room is added, **Houses** gain an additional window. If a **House** has more than 3 floors, the top floor gains 2 windows per room. For example, if a **House** has 2 floors and 3 rooms, it has (2 x 2) + (3 x 1) = 7 windows.

Add a **toString** method which outputs a **String** of the form:
```
House: n Windows, m Rooms, Floor Space: o, and p
Floors
```
where n, m, o, and p have the appropriate values for a given **House**.

Do not permit negative input parameters anywhere in the class.

The unit test case for this question is called **testHouse**. This test will:

i. Construct a House with 2 floors
ii. Add three rooms
iii. Check that each method in the interface returns the correct result
iv. Call the **toString** method and check it is correct

(c) (3 marks)
Create a class called **Bungalow** which inherits the methods of **House**.

Bungalows only have 1 floor, and as such only require a default constructor, and should not permit accessor or mutator methods to reference floors other than 0 (your implementation from part b may potentially do this already).

Modify the inherited **toString** method appropriately (if needed). For example:
```
Bungalow: 4 windows, 2 Rooms, Floor Space: 40.0, and 1
Floor
```

Enforce that **Bungalows** cannot have a floor space larger than 60 in all appropriate places of the class.

The unit test case for this question is called **testBungalow**. This test will:

i. Construct a Bungalow
ii. Add two rooms
iii. Check that each method in the interface returns the correct result
iv. Call the **toString** method and check it is correct

(d) (8 marks)
Create a class called **Estate**, its instance variables are:
- An **ArrayList** of **Buildings:** called **buildings**
- A **double** to represent a policy on garden size: called **gardenSizePolicy**
- An **int** to represent the maximum permitted estate area: called **maxEstateArea**

Make one constructor for all variables (in the order below) with the following constraints:
- if **ArrayList** is null, construct a new one
- **gardenSizePolicy** must be in the range 0.5 – 3 inclusive, and
- **maxEstateArea** must be positive, but may never be exceeded anywhere in the class, i.e. when attempting to add a building (see below) this value cannot be exceeded

The class should also have the following methods:
1. **addBuilding(Building b)**: adds a new Building (allows duplicates) but must inhibit the **maxEstateArea** from being exceeded
2. **calculateArea** calculates the total area of the estate as a **double**. (You can modify the classes or interface as you need to answer this: add a short comment explaining your rationale in the code).
   The area that a **Building** occupies is based on its largest floor, and the size of garden it has. This is calculated as follows:

   House area = largest floor + (**gardenSizePolicy** x largest floor)
   Bungalow area = largest floor + ((**gardenSizePolicy** / 2) x largest floor)

   The **Estate** area is the sum of all **House** and **Bungalow** areas in the **Estate**.

3. **getNumWindows** returns (as an **int**) the total number of windows in the **Estate**, i.e. the sum of windows in all **Buildings**

The unit test case for this question is called **testEstate**. This test will:

i. Construct an **Estate** with input variables **null** for the **ArrayList**, 2 for **gardenSizePolicy**, and 160 for **maxEstateArea.**
ii. The test will then add a **House** and a **Bungalow** via the method **addBuilding(Building b).**
iii. The test will check that the method **calculateArea** returns the correct result after adding each **Building**.
iv. The test will check that the method **getNumWindows** returns the correct result.
v. Finally, the test will add the **Bungalow** again (this will exceed the **maxEstateArea**) and check that an error occurs.
vi. This unit test has comments to explain how the areas and number of windows should be calculated.

(e) (2 x 5 = 10 Marks)
Answer any **TWO** of the video recording questions (see separate PDF, which will be released at 10:45). This question is a MUST PASS question, i.e. a score of at least 4 out of 10 is needed to pass the exam.

(f) (7 marks)
Ensure that your code is:

i. well formatted **(0.5 marks)**
ii. applies any appropriate best practices and/or minimises repetition **(5.5 marks)**
iii. appropriately commented **(1 mark)**

--- end of Question 1 ---

Question 2

Answer parts (a) to (c) with solutions coded in Java, and answer part (d) via video recording and demonstration.

35 marks in total

In this question, you will make **one** interface, and at least **one** class (depending on how you answer the question):

1. an interface called **Stringifier** in part a
2. a class called **StringifyException** in part b
3. complete the class **Main** in part c (you may also create additional classes depending on how you answer the question)

Note: there are multiple ways to answer this question.

The unit tests for this question are located in **TestsQuestionTwo.java**

(a) (1 mark)
Create an interface **Stringifier** that has only one method: **stringify** which returns an **ArrayList** of **Strings** and takes an **Object** array as input.

The unit test case for this question is **testInterface.** It checks that the method has the correct signature.

(b) (2 marks)
Create a class **StringifyException** (a **checked Exception**) which will be thrown if specific conditions of **stringify** are not met in part c.

The unit test for this question is **testException.** It checks that the class is some form of **Exception**, i.e. that it is **Throwable**.

(c) (15 marks)
Complete the class **Main** so that the instance variable implementations of **Stringifier** (i.e., their **stringify** method) correspond to the definitions below.

In all cases, the **stringify** method should throw a **StringifyException** if the array is empty.

Answer this question in the manner you are most comfortable with, but pay attention to opportunities for code reuse.

1. **concatenator (2 marks):** creates and returns a **String** that concatenates a **String** representation of all elements in the **Object array.** These should be separated by a space and have a full stop at the end – there will only ever be one item in the returned **ArrayList:** the concatenated **String.**

   Example:
   The **Object array** has the values:
   i. a **Double** with the value 1
   ii. an **Integer** with the value 2
   iii. a **Float** with the value 3
   iv. a **String** with the value "4"

   This should return an **ArrayList<String>** with one element with the value:
   `1.0 2 3.0 4.`

   The unit test case for this question is called **testConcatenator.** It tests the above example.

2. **translator (6 marks):** transforms the input into a natural language representation of the mathematical expression(s) provided – these will always be **Strings.** A **StringifyException** should be thrown if any element in the **Object array** is not a **String.** The **ArrayList** will have one expression per **String**, i.e. if the **length** of the **array** is 2, there are two expressions.

   The only operations will be: +, -, x and / (plus, minus, times, and divided by) and the answer will always be in the range of 0 – 10 (inclusive). There is, however, **no limit** to the length of the expression. You **do not** need to check that the answer is correct. Be careful with + (plus) if using a regular expression-based approach (this question can be answered without regular expressions).

Example 1: 1 + 1 = 2
Is translated to: `one plus one equals two`

Example 2: 2 x 3 + 1 = 7
Is translated to: `two times three plus one equals seven`

Example 3:
The **Object array** has the values:
i. a **String** with the value: 1 + 1 = 2
ii. a **String** with the value: 2 x 3 + 1 = 7

This should return an **ArrayList<String>** with two elements with the values:
```
one plus one equals two
two times three plus one equals seven
```

The unit test case for this question is called **testTranslator.** It tests the above examples.

3. **sumUp (4 marks):** operates on either all **Strings** or all **numbers** (will only be **Integers** or **Doubles**).

   If all **Strings**, concatenate them with a space between each **String** and a full stop at the end.

   If all **numbers**, find their sum with **double** precision and return this as a **String**.

   The **ArrayList** will only have one item: the concatenated **String**, or a **String** representation of the sum of numbers.

   Throw a **StringifyException** if the array is not all **Strings** or all **numbers (Integers** or **Doubles**).

   Example 1 (all Strings):
   The **Object array** has the values:
   i. "this"
   ii. "is"
   iii. "an"
   iv. "array"
   v. "of"
   vi. "Strings"

   This should return an **ArrayList<String>** with one element with the value:
   ```
   this is an array of Strings.
   ```

Example 2 (all numbers):
The **Object array** has the values:

i. A **Double** with the value 1
ii. An **Integer** with the value 2
iii. A **Double** with the value 3

This should return an **ArrayList<String>** with one element with the value:
```
6.0
```

The unit test case for this question is called **testSumUp.** It tests the above examples.

4. **reverser (3 marks):** checks whether the **concatenator** produces the same output (ignoring the full stop) when the array is reversed. If so, the only item in the **ArrayList** is the **String** "true" and "false" otherwise.

   Example 1 (returns true):
   The **Object array** has the values:

   i. An **Integer** with the value: 1
   ii. A **String** with the value: 6
   iii. An **Integer** with the value: 1

   This should return an **ArrayList<String>** with one element with the value:
   ```
   true
   ```

   Example 2 (returns false):
   The **Object array** has the values:

   i. A **String** with the value: A
   ii. A **String** with the value: B
   iii. A **String** with the value: C

   This should return an **ArrayList<String>** with one element with the value:
   ```
   false
   ```

   The unit test case for this question is called **testReverser.** It tests the above examples.

(d) (2 x 5 = 10 Marks)
Answer any **TWO** of the video recording questions (see separate PDF, which will be released at 10:45). This question is a MUST PASS question, i.e. a score of at least 4 out of 10 is needed to pass the exam.

(e) (7 marks)
Ensure that your code is:

iv. well formatted
(0.5 marks)
v. applies any appropriate best practices and/or minimises repetition
(5.5 marks)
vi. appropriately commented
(1 mark)

--- end of Question 2 ---

--- end of Part A ---

University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

AUTUMN TRIMESTER EXAMINATIONS

ACADEMIC YEAR 2020/2021

COMP20250: Introduction to Java

COMP20300: Java Programming (Mixed Delivery)

Exam Part A: Video Recording Questions

Assoc Prof. Chris Bleakley
Dr. Simon Caton

Time Allowed: 45 mins

Answer either Question 1 or Question 2.
This choice should MATCH the main paper.

Question 1

Answer parts (a) to (d) with solutions coded in Java, and answer part (e) via video recording and demonstration.

35 marks in total

See main paper for parts (a)-(d) and (f)

(e) (2 x 5 = 10 Marks)

Answer any **TWO** of the following four questions by recording your screen, pointing at / referring to specific parts of your code, and orally presenting your answer. **You do not need slides or any other presentation materials.**

This is a **MUST PASS** question, i.e. you must score 4 / 10 or more in this question.

Notes:

- You may upload 1 video per question, or one video for both questions.
- Make sure your code font size is large enough to view in the video.
- The total recording time for both questions, should not (significantly) exceed 5 mins.

Record yourself questions for Question 1:

1- Explain your rationale for using (or not using) **two** of the following reserved words, and use examples in your code to justify why you did or didn't use them:

   a. static (do not discuss a main method),

   b. super,

   c. protected,

   d. final

2- Which of the unit test(s) (that your code passes) was/were the most helpful in developing your solution and discuss how it/they helped you answer the question(s)?

3- Pick an example of polymorphism in your code and explain how the use of polymorphism improves your code quality.

4- How would you change the **addBuilding(Building b)** method (and any other associated parts of the code) to prevent duplicates, i.e. when 2 **Buildings** are the same?

--- end of Question 1 ---

Question 2

Answer parts (a) to (c) with solutions coded in Java, and answer part (d) via video recording and demonstration.

35 marks in total

See main paper for parts (a)-(c) and (e)

Question 2(d) (2 x 5 = 10 Marks)

Answer any **TWO** of the following four questions by recording your screen, pointing at / referring to specific parts of your code, and orally presenting your answer. **You do not need slides or any other presentation materials.**

This is a **MUST PASS** question, i.e. you must score 4 / 10 or more in this question.

Notes:

- You may upload 1 video per question, or one video for both questions.
- Make sure your code font size is large enough to view in the video.
- The total recording time for both questions, should not (significantly) exceed 5 mins.

Record yourself questions for Question 2:

1- Which of the unit test(s) (that your code passes) was/were the most helpful in developing your solution and discuss how it/they helped you answer the question(s)?
2- How did you minimise code repetition in part (c)? If you didn't do this, discuss how you would do it now. In either case, draw attention to specific parts of the code where you have (or could have) minimised code repetition.
3- Reflect on your decision(s) for designing some of your solutions in part (c). Here, you should focus on whether you needed to make more classes, or if you included all code directly in the class **Main**. Do you think your approach was the "best" way you could have answered this question, or are there other "better" ways? Include (briefly) why you think this.
4- Do you agree with the use of a **checked Exception** in part (b)? If so why, if not, why not?

--- end of Question 2 ---

University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

AUTUMN TRIMESTER EXAMINATIONS

ACADEMIC YEAR 2020/2021

COMP20250: Introduction to Java

COMP20300: Java Programming (Mixed Delivery)

Exam Part B

Assoc Prof. Chris Bleakley
Dr. Simon Caton

Time Allowed: 1.5 Hours (+30 min for video recording)

Answer one question, i.e., question 3, question 4, or question 5.

The ExamResourcesPartB.zip file has the following contents:

- **TestsQuestionThree.java**, which contains the test cases for question 3
- **TestsQuestionFour.java**, which contains the test cases for question 4
- **TestsQuestionFive.java**, which contains the test cases for question 5
- **numbers.txt**, a file used in question 5 parts (c) – (e)
- **integers.txt**, a file used in question 5 part (f)

<u>NOTE</u>**: ExamResourcesPartB.zip** also contains JUnit 4 versions of the tests. **These can be ignored by the majority of the class**. They are there only for students that reported problems with JUnit 5 in part A. They follow the same naming convention as the test files above, and have the same test cases, but use JUnit 4 assertions. They have file names ending **JU4.java**.

Summary of questions:

- **Question 3** focuses on outputting two String-based shapes according to specific input parameters, e.g.:

```
+++++++
+ooooo+
+o+++o+
+o+o+o+
+o+++o+
+ooooo+
+++++++
```

```
|||||||||||||||
|*************|
|*|||||||||||*|
|*|*********|*|
|*|*********|*|
|*|||||||||||*|
|*************|
|||||||||||||||
```

```
+++++ooooo+++++
+++++ooooo+++++
+++++ooooo+++++
+++++ooooo+++++
+++++ooooo+++++
```

```
---------------
---------------
---------------
|||||||||||||||
|||||||||||||||
|||||||||||||||
---------------
---------------
---------------
```

- **Question 4** (starts on p. 5) focuses on array and ArrayList manipulation, with elements of casting and method overloading
- **Question 5** (starts on p. 8) focuses on reading text files and isolating parts of the content

Question 3

Answer parts (a) to (c) with solutions coded in Java, and answer part (d) via video recording and demonstration.

35 marks in total

In this question, you will create one class called **Draw.**
The unit tests for this question are in **TestsQuestionThree.java**

Hint: for this question focus on iteratively building up towards the desired pattern.

(a) (4 marks)

Define a class called **Draw**, with 2 class methods: one called **boxes**, and one called **stripes.** These will both output a **String** pattern.

Both methods take 2 **int** and 2 **char** parameters. The first **int** is the height of the pattern, the second **int** is the width.

The method **stripes** also takes a fifth parameter which is a **boolean** representing the direction of the stripes in the pattern: **true** means they should be vertical, **false** means they should be horizontal.

Ensure that the following condition is maintained for both methods:

- The **char** parameters cannot be equal

Ensure that the following condition is maintained for the method **boxes:**

- The width and height must always be in the range 7-20 (inclusive)

Ensure that the following conditions are maintained for the method **stripes:**

- If the **boolean** is false (i.e. the stripes are vertical), the width must be a multiple of three
- If the **boolean** is true (i.e. the stripes are horizontal), the height must be a multiple of three
- The width and height must always be 5-25 (inclusive)

The test case for this question is called: **testPartA**. It tests:

i. the signature of the method, and
ii. whether an error is thrown in response to invalid input

(b) (8 marks)
Implement the method **boxes** such that it returns a **String** that contains a pattern of a box within a box within a box. See examples overleaf.

Example 1:
```
String s = Draw.boxes(7,7,'+', 'o');
System.out.println(s);

+++++++
+ooooo+
+o+++o+
+o+o+o+
+o+++o+
+ooooo+
+++++++
```

Example 2:
```
String s = Draw.boxes(8,15,'|', '*');
System.out.println(s);
|||||||||||||||
|*************|
|*|||||||||||*|
|*|*********|*|
|*|*********|*|
|*|||||||||||*|
|*************|
|||||||||||||||
```

Essentially, the first **char** represents the outer box and is present on the outside of the pattern. The second **char** is placed inside (drawing another box), a final box is drawn with the first **char**, and then all remaining locations in the pattern take the value of the second **char**.

Hint: the 2nd box can be viewed as not being part of either the 1st or 3rd box, the middle can also be viewed this way. In other words: the *s in Example 2 do not belong to either box 1 or box 3. So you would only need to identify when to place the first **char**, as in all other situations you place the second **char.**

The test case for this question is called: **testPartB**, it tests the two examples above.

(c) (8 marks)
Implement the method **stripe** such that it returns a **String** that contains a pattern of a three parallel stripes of equal width / height. See examples below. The first **char** is always stripe 1 and 2, and the second **char** is always the middle stripe.

Example 1 (vertical stripes):
```
String s = Draw.stripes(5, 15,'+', 'o', false);
System.out.println(s);

+++++ooooo+++++
+++++ooooo+++++
+++++ooooo+++++
+++++ooooo+++++
+++++ooooo+++++
```

Example 2 (horizontal stripes):

```
String s = Draw.stripes(9,15,'-', '|', true);
System.out.println(s);
```

```
---------------
---------------
---------------
|||||||||||||||
|||||||||||||||
|||||||||||||||
---------------
---------------
---------------
```

The test case for this question is **testPartC**, it tests the two above examples.

(d) (2 x 5 = 10 Marks)
Answer any **TWO** of the video recording questions (see separate PDF, which will be released at 10:45). This question is a MUST PASS question, i.e. a score of at least 4 out of 10 is needed to pass the exam.

(e) (5 marks)
Ensure that your code is:

i. well formatted

(0.5 marks)

ii. applies any appropriate best practices and/or minimises repetition

(3.5 marks)

iii. appropriately commented

(1 mark)

--- end of Question 3 ---

Question 4

Answer parts (a) to (f) with solutions coded in Java, and answer part (g) via video recording and demonstration.

35 marks in total

In this question, you will create one class called **Question4**
The unit tests for this question are in **TestsQuestionFour.java**

(a) (2 marks)
Create a constructor that initialises two instance variables: 1) an **ArrayList** of type **String** called **arrayList**, and 2) a **double** array (not an **ArrayList**!) called **doubleArray**. The constructor should have one input parameter, an **int** which defines the initial size of the **double** array (this should be positive).

Add accessor methods for these two instance variables.

The test case for this question is called **testPartA**, it tests:

i. the constructor, and
ii. whether the two accessor methods return what it expected, and are correctly named

(b) (6 marks)
Create as many mutator methods with the name **add** as you need to be able to handle receiving **ints**, **doubles floats**, **Strings** and **Objects**. Numeric data (i.e. **int**, **double**, and **float**) should be inserted into the **double** array. Everything else should be added to the **ArrayList**. When an **Object** is received, call its **toString** method and add the **String** to the **ArrayList**.

Note: If the size of the **double** array is not large enough to store new values, you should resize it so that its length increases by 5.

Note: you do not need to check if a **String** is a number. For example, if a **String** were to have the value “2” it would still go in the **ArrayList**.

The test case for this question is called: **testPartB**. It has the following functionality:

i. Calls the add method with the following inputs: an **int (1), float (2), double (3), String (“4”), String (“5”),** and **Double (6)**
ii. Calls the two accessor methods, and verifies that the content of the **double** array and **ArrayList** are correct.
iii. It then forces the **double** array to resize (5 was passed to the constructor), by adding 3 additional **int**s.
iv. Finally, it checks that the **double** array was correctly resized, and that all elements in the array are correct

(c) (2 marks)
Add a **toString** method, which returns all elements in the **ArrayList**, followed by all elements in the **double** array as a comma separated list. For example, if the **ArrayList** contains "Java" and "Exam" and the **double** array contains 1 and 2, the **toString** method should return "Java, Exam, 1.0, 2.0".

Note: do not return empty values in the **double** array in the **toString** method.

Example:

i. Add “Java”
ii. Add “Exam”
iii. Add 1
iv. Add 2

toString should return:

```
Java, Exam, 1.0, 2.0
```

The test case for this question is called **testPartC**. It tests the above example.

(d) (3 marks)
Add as many methods with the name **contains** as you need which all return **true** if the passed value is present in either the **ArrayList** or **double** array, and **false** if it is in neither. The only types you need to handle are: **ints**, **doubles floats**, **Strings** and **Objects.**

The test case for this question is called **testPartD**. It has the following functionality:

i. Adds an **int (1), double (2), String ("Java"), and a String ("Exam")**
ii. Calls contains for each of these values and checks that the method returns **true**
iii. Calls contains with the input: **int (3), String ("Java Exam"), and Double (2)** and checks that the method returns **false**

(e) (4 marks)

Create an add method that receives an **ArrayList<Object>** as input. This will contain numeric **Objects** (**Double**, **Float**, **Integer** etc.) and non-numeric **Objects**. Cast/convert appropriately (and if needed), and then add all elements to either the **ArrayList<String>**, or for numeric data the **double** array.

The test case for this question is called **testPartE**. It has the following functionality:

i. Constructs an **ArrayList<Object>** with the values a **Double (2), Integer(3), Float(4), String ("5")** and **String ("6")**
ii. Calls the **add** method passing the **ArrayList<Object>**
iii. Checks that the contents of the **double array** are the first three values in the **ArrayList<Object>**
iv. Checks that the contents of the **ArrayList<String>** are the last two values in the **ArrayList<Object>**

(f) (3 marks)

Create a method called **reverseAndDuplicate**. This method should reverse the order of the **ArrayList** and **double** array, and add its contents (in the new order) again.

Example:

i. Add "Java"
ii. Add "Exam"
iii. Add 1
iv. Add 2

The contents of the:

- **double array** are: 1.0 and 2.0 in this order, and
- the contents of the **ArrayList<String>** are: “Java” and “Exam”, in this order

After calling **reverseAndDuplicate**, the contents of the:

- **double array** are: 2.0, 1.0, 2.0, and 1.0 in this order, and
- the contents of the **ArrayList<String>** are: “Exam”, “Java”, “Exam”, and “Java” in this order

The test case for this question is called **testPartE**. It tests the above example.

(g) (2 x 5 = 10 Marks)
Answer any **TWO** of the video recording questions (see separate PDF, which will be released at 10:45). This question is a MUST PASS question, i.e. a score of at least 4 out of 10 is needed to pass the exam.

(h) (5 marks)
Ensure that your code is:

i. well formatted

(0.5 marks)

ii. applies any appropriate best practices and/or minimises repetition

(3.5 marks)

iii. appropriately commented

(1 mark)

--- end of Question 4 ---

Question 5

Answer parts (a) to (f) with solutions coded in Java, and answer part (g) via video recording and demonstration.

35 marks in total

In this question, you will create one class called **Question5**
The tests for this question are in **TestsQuestionFive.java**

NOTE you need to modify lines 12 and 13 in **TestsQuestionFive.java** to reflect where you have saved the test files for this question. Remember on Windows, you need to escape a backslash in a file path, i.e. “\” should be “\\”.

(a) (2 marks)
Create three **int** constants:

1- FIRST_TOKEN
2- LAST_TOKEN
3- ITH_TOKEN

Create an **enum** called **operator** with the values:

1. **SUM**
2. **PRODUCT**

The test case for this question is called: **testPartA**. It checks that:

i. the **constants** are correctly named, and
ii. that the **enum** and its values are correctly named.

(b) (1 mark)
Create a class method called **readFile** that receives a **String** (this will be an absolute path to a file which you can use to construct a **File** object), a **char** which represents a separator (for tokenising a **String**), and an **int** (which should be one of the constants declared in part a).

The method returns a **String**.
You do not need to do any **IOException** handling in the method itself: the method signature should reflect this.

The file will always be a text file. However, the separator used to distinguish words will vary. The **char** input parameter defines how words are separated.

For example:
`This-is-a-sentence` has words separated by a hyphen: '-'
`This@is@a@sentence` has words separated by an at: '@'
`This is a sentence` has words separated by a space: ' '

Thus, you will need to use the value of the **char** input variable to split the text. All examples files have words separated by a space ' '.

The test case for this question is called: **testPartB**. It checks that the **readFile** method signature is correct.

Questions (c) – (e) define the behaviour of the method in response to the three **int** constant values defined in part (a). These questions use the file **"numbers.txt"**.

(c) (2 marks)
Your **readFile** method should handle **FIRST_TOKEN**: here you should return the first word of each line concatenated into a single **String** and separated by a space.

Example (char = ' ')
Line 1 of the file is: one two three
Line 2 of the file is: four five six
Line 3 of the file is: seven eight nine
Line 4 of the file is: ten eleven twelve

The output should be:
```
one four seven ten
```

The test case for this question is called: **testPartC**. It tests the above example using the file **"numbers.txt"**

(d) (3 marks)

Your **readFile** method should handle **LAST_TOKEN**: this is the same as the previous question, however instead of using the first word, it should use the last word.

Example: (char = ' ')
Line 1 of the file is: one two three
Line 2 of the file is: four five six
Line 3 of the file is: seven eight nine
Line 4 of the file is: ten eleven twelve

The output should be:
```
three six nine twelve
```

The test case for this question is called: **testPartD**. It tests the above example using the file **"numbers.txt"**

(e) (4 marks)
Your **readFile** method should handle **ITH_TOKEN**. In this setting, the method should use the 1st word from the first line, the 2nd word from the second line, the third word from the third line, etc. i.e. it uses the ith word, where "i" is the current line number.

If the line does not contain sufficient words, for example the fourth line only contains 3 words, the method should insert "ERROR" for this line, but continue reading the file.

Example: (char = ' ')
Line 1 of the file is: one two three
Line 2 of the file is: four five six
Line 3 of the file is: seven eight nine
Line 4 of the file is: ten eleven twelve

The output should be:
```
one five nine ERROR
```

The test case for this question is called: **testPartD**. It tests the above example using the file **"numbers.txt"**

(f) (8 marks)
Make a new class method called **readFileNumeric**. **readFileNumeric** should have the same signature as **readFile**, but take a 4th parameter of type **operator** (the **enum** declared in part a) and have a return type of **int** instead of **String.**

The method will be instructed to read text files that contain lines of positive integers separated by the value of the **char** input parameter. The functionality you implemented in parts (c)-(e) remains the same (as all integers are read as **Strings** initially):
- For **FIRST_TOKEN** operates on the first integer of each line
- For **LAST_TOKEN** operates on the last integer of each line

- For **ITH_TOKEN** operates on the ith integer of each line – in cases where there is not an ith token, throw an appropriate error instead

As such, you should decide how you want to reuse your solution for parts (c)-(e) in this question. **Note** the files passed to this method will only ever have integers in them, you **do not** need to control for bad input.

For the integers you read, sum them if the **enum** has the value **SUM**, and multiply them together if the **enum** has the value **PRODUCT**, and return the result.

Example 1 (char = ' ', int is **LAST_TOKEN**, operator is SUM)
Line 1 of the file is: 1 2 3
Line 2 of the file is: 4 5 6
Line 3 of the file is: 7 8 9
Line 4 of the file is: 10 11 12

The output should be (3 + 6 + 9 + 12):
```
30
```

Example 2 (char = ' ', int is **FIRST_TOKEN**, operator is PRODUCT)
Line 1 of the file is: 1 2 3
Line 2 of the file is: 4 5 6
Line 3 of the file is: 7 8 9
Line 4 of the file is: 10 11 12

The output should be (1 x 4 x 7 x 10):
```
280
```

The test case for this question is **testPartF**. It tests the above two examples using the file **"integers.txt"**

(g) (2 x 5 = 10 Marks)
Answer any **TWO** of the video recording questions (see separate PDF, which will be released at 10:45). This question is a MUST PASS question, i.e. a score of at least 4 out of 10 is needed to pass the exam.

(h) (5 marks)
Ensure that your code is:

iv. well formatted
(0.5 marks)

v. applies any appropriate best practices and/or minimises repetition
(3.5 marks)

vi. appropriately commented
(1 mark)

--- end of Question 5 ---

--- end of Part B ---

University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

AUTUMN TRIMESTER EXAMINATIONS

ACADEMIC YEAR 2020/2021

COMP20250: Introduction to Java

COMP20300: Java Programming (Mixed Delivery)

Exam Part B: Video Recording Questions

Assoc Prof. Chris Bleakley

Dr. Simon Caton

Time Allowed: 45 mins

Answer one question, i.e., question 3, question 4, or question 5.

This choice must MATCH the main paper.

Question 3

Answer parts (a) to (c) with solutions coded in Java, and answer part (d) via video recording and demonstration.

35 marks in total

See main paper for parts (a)-(c) and (e)

Question 3(d) (2 x 5 = 10 Marks)

Answer any **TWO** of the following four questions by recording your screen, pointing at / referring to specific parts of your code, and orally presenting your answer. **You do not need slides or any other presentation materials.**

This is a **MUST PASS** question, i.e. you must score 4 / 10 or more in this question.

Notes:

- You may upload 1 video per question, or one video for both questions.
- Make sure your code font size is large enough to view in the video.
- The total recording time for both questions, should not (significantly) exceed 5 mins.

Record yourself questions for Question 3:

1- Explain how you built up your solution for **part (b),** i.e. what steps did you go through to create the pattern?

2- Explain how you built up your solution for **part (c),** i.e. what steps did you go through to create the pattern?

3- Explain how you would modify your answer **part (b),** to have an additional inner box.

4- Explain how you would modify your answer to **part (c),** to have an additional parameter that would allow you to control the number of stripes.

--- end of Question 3 ---

Question 4

Answer parts (a) to (f) with solutions coded in Java, and answer part (g) via video recording and demonstration.

35 marks in total

See main paper for parts (a)-(f) and (h)

Question 4(g) (2 x 5 = 10 Marks)

Answer any **TWO** of the following four questions by recording your screen, pointing at / referring to specific parts of your code, and orally presenting your answer. **You do not need slides or any other presentation materials.**

This is a **MUST PASS** question, i.e. you must score 4 / 10 or more in this question.

Notes:

- You may upload 1 video per question, or one video for both questions.
- Make sure your code font size is large enough to view in the video.
- The total recording time for both questions, should not (significantly) exceed 5 mins.

Record yourself questions for Question 4:

1- Which of the unit test(s) (that your code passes) was/were the most helpful in developing your solution? Discuss how it/they helped you answer the question(s).
2- Discuss your strategy for minimising repetition for either the **add** or **contains** methods. If you did not do this, explain how (if you had more time) you would go about minimising code repetition.
3- Discuss how you used (or if you did not do so, could use) **any two** of the following, using your code as a point of reference and examples:
   a. Autoboxing and/or unboxing
   b. Casting
   c. instanceof
   d. Any method in the System class (**except methods that print to the terminal**)
4- Discuss your approach to resizing the **double array**: why did you do it this way, and what alternative approaches did you consider?

--- end of Question 4 ---

Question 5

Answer parts (a) to (f) with solutions coded in Java, and answer part (g) via video recording and demonstration.

35 marks in total

See main paper for parts (a)-(f) and (h)

Question 5(g) (2 x 5 = 10 Marks)

Answer any **TWO** of the following four questions by recording your screen, pointing at / referring to specific parts of your code, and orally presenting your answer. **You do not need slides or any other presentation materials.**

This is a **MUST PASS** question, i.e. you must score 4 / 10 or more in this question.

Notes:

- You may upload 1 video per question, or one video for both questions.
- Make sure your code font size is large enough to view in the video.
- The total recording time for both questions, should not (significantly) exceed 5 mins.

Record yourself questions for Question 5:

1- Which of the unit test(s) (that your code passes) was/were the most helpful in developing your solution? Discuss how it/they helped you answer the question(s).
2- How did you reuse your code from parts (c) – (e) in part (f)? If you didn't (or didn't do this question) how would you have reused it?
3- Discuss your strategy for reading lines of text and splitting the lines into words. Did you consider other options? If so, what were they and why did you not use them?
4- Did you prefer working with the constants or the enum? Explain your choice.

--- end of Question 5 ---

--- end of Part B ---