Programming and Systems Development Final Lab Exam – 2021

During the exam

You are not allowed to access any previously written documents or code files from your home drive or the web. If you do, it will be considered plagiarism.

As in a normal examination, communication between candidates during the laboratory exam is strictly forbidden.

Any candidate who experiences a hardware or software problem during the examination should summon an invigilator at once.

Overview

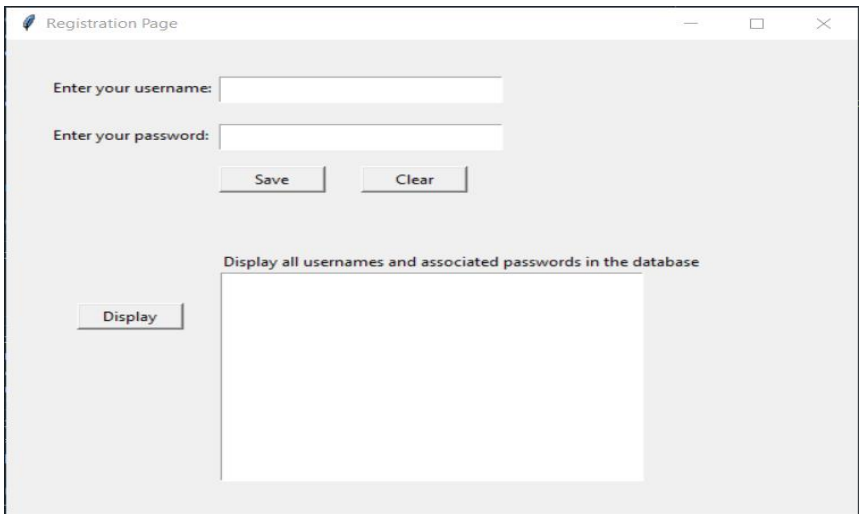
This exam consists of two parts: part 1 requires you to write a Python program, and part 2 requires you to write a Java program. The instructions, allocated marks, and submission instructions are given below for each part. Note that each part has a total of **24 marks** allocated for correctness, as indicated; in addition, **1 mark** for each part will be allocated for appropriate coding style (commenting, formatting, variable names, etc).

Part 1: Python

You should write the code for each task below in the same **python source file**, as indicated below. You should then submit **your source file** through Moodle as described at the end.

Python Task 1a: tkinter GUI and Database

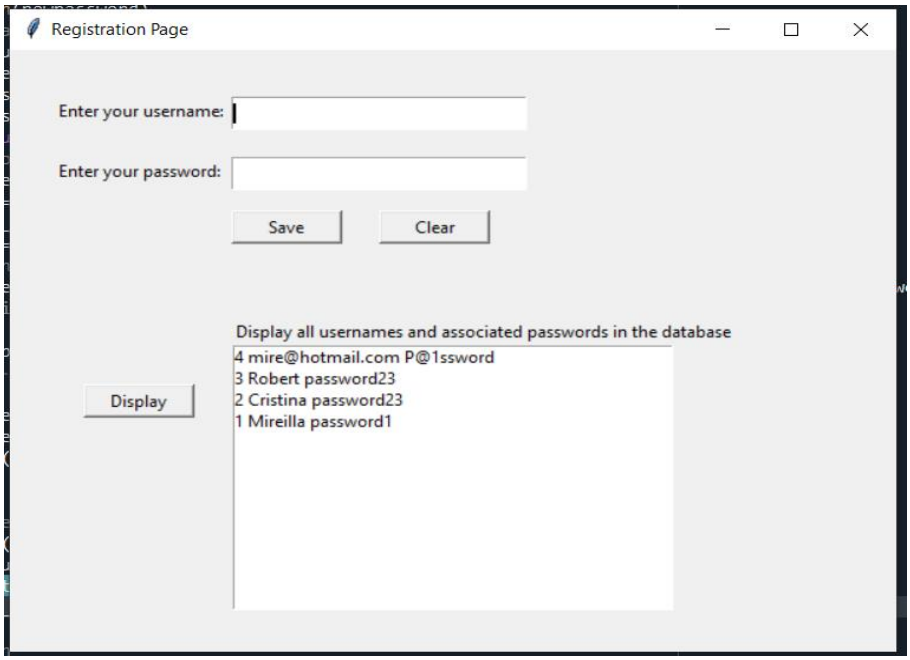
Create a program that should display the following screen **[4 marks]:**



The programme should save the valid usernames and passwords in a database called **Users** when the Save button is clicked. **[4 marks]**

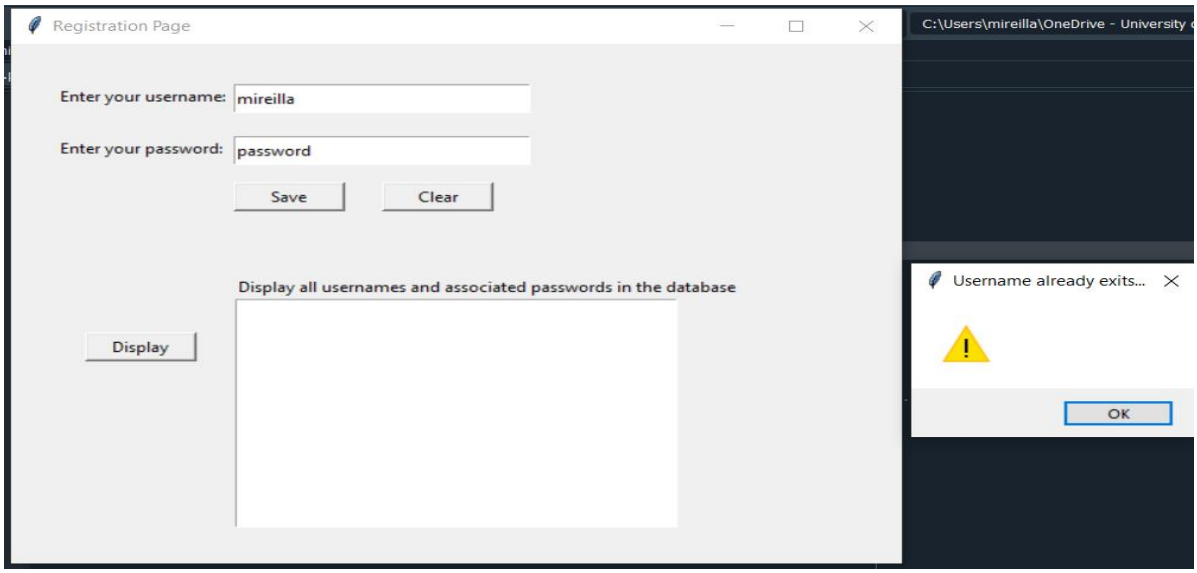
It should clear all the windows when the Clear button is clicked. **[2 marks]**

If a user clicks the button “Display”, the system should display all usernames and passwords saved in the database. **[3 marks]**



Python Task 1b: Username and password validation

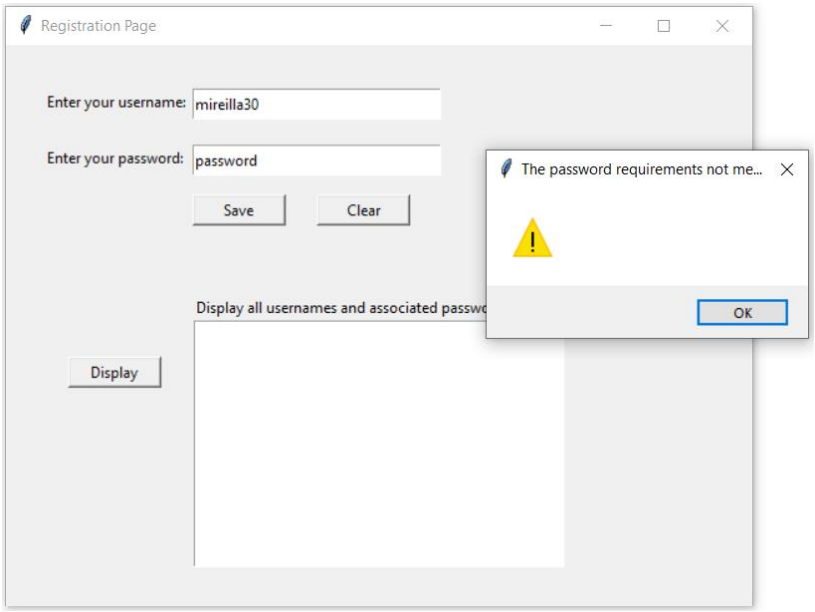
The system should not save duplicated usernames. If a user enters a username, the system should check if the username already exists in the database. If the username already exists in the database, the system should display a suitable message**. [5 marks]**



The password should meet the following criteria:

a) The password should have a minimum of 8 characters.
b) The password should include numbers.

If the password does not meet BOTH criteria, the system should reject it with a message saying that it does not meet the requirements. **[6 marks]**



Python submission

Ensure your submission file is named exactly as it is supposed to (i.e. **your_name_python_task.py)**

Go to your moodle account, and submit your file at the appropriate submission link:

Moodle > Programming and System Design > Lab Exam > Python Submission

Part 2: Java

Your task is to develop classes to model a world involving **Citizens** and **Traders**, where **Citizens** are able to exchange gems for **Goods** including bread, wool, armour, weapons, and building materials. All of those goods are provided by **Traders** in exchange for gems. Every **Trader** supports one or more **Trades**, where a **Trade** includes the following information:

- The price in gems
- The specific goods that are for sale
- How much of that goods is available for the price

The following is an example of a **Trade**:

- Trade **1 gem** for **3 Bread**

Every **Citizen** has an amount of gems along with an amount of Goods of each type (which may be zero). When a **Citizen** successfully executes a **Trade**, the following happens:

- The relevant amount of gems is removed
- The relevant amount of Goods is added

For example, if a **Citizen** initially has 5 gems and an empty inventory, after executing the above trade, they would have 4 gems and 3 Bread in their inventory. If a **Citizen** does not have enough gems to complete a trade, they cannot execute it.

Trades are provided by **Traders**. When a new **Trader** is created, they only have one Trade available. Each time a **Trader** makes a trade with a **Citizen**, a new (randomly-chosen) **Trade** is added to their set. So if a **Trader** starts with the sample **Trade** above, after three successful trades with **Citizens**, the **Trader** might have the following **Trades** in their list:

- Trade **1 gem** for **3 Bread**
- Trade **2 gems** for **1 Helmet**
- Trade **1 gem** for **1 Bread**
- Trade **2 gems** for **4 Wool**

(Note that this trader has two **Trades** for the same item of **Goods** – Bread. This is not a problem in the system – you do not need to check for duplicates.)

Java Task 1: Goods (3 marks)

*Note about implementation: all classes created in this exam should be put in the **trading** package.*

You must create an enumerated type **Goods** with the following values:

BREAD, COAL, FISH, HELMET, IRON, PAPER, SHIELD, SWORD, WOOD, WOOL

Java Task 2: Trade (7 marks)

You must create a class **Trade** representing a single trade, including the following properties:

- gems: the number of gems involved in the trade (an integer)
- amount: the amount of goods involved in the trade (an integer)
- goods: the type of goods involved in the trade (an object of type Goods)

For example, the trade "1 gem for 3 BREAD" would be represented as:

- gems: 1
- amount: 3
- goods: Goods.BREAD

The **Trade** class should have a public constructor that initialises all of the fields, and should also include the following methods:

- A complete set of **get** methods, but no **set** methods
- Appropriate implementations of **equals()** and **hashCode()** – note that equality should be based on the values of all three fields.
- An implementation of **toString()** that produces a string representation of the **Trade** similar to the example above ("1 gem for 3 BREAD").

Not that you can use automatically-generated code for the **get** methods and **equals()**/**hashCode()**, but you will need to write the **toString()** method by hand to meet the specification.

Java Task 3: Citizen (6 marks)

Next, create a class **Citizen** representing a citizen in the game. The internal details of this class are up to you; here is the required behaviour.

The constructor for **Citizen** should take a single parameter, an integer representing the number of gems, and should create a new **Citizen** with that many gems and an empty inventory.

Citizen should have the following **public** methods:

- **public int getGems()** – returns the current amount of gems
- **public int getAmount (Goods goods)** – returns the current amount of the indicated **Goods** type in the inventory. Should return 0 if the **Citizen** does not have any of the indicate **Goods.**
- **public boolean executeTrade (Trade trade)** – should check whether the given trade is possible (i.e., whether the **Citizen** has enough gems)
    - If the amount of gems is not enough, **return false** and do not change anything
    - Otherwise, update the amount of gems and the inventory based on the details of the Trade and **return true**

Java Task 4: Trader (4 marks)

Create a class **Trader** representing a trader in the world. As with **Citizen**, the internal details of this class are up to you; here is the required behaviour.

The constructor for **Trader** should take no parameters, and should create a **Trader** with a single, randomly-chosen **Trade**.

Trader should have the following **public** methods:

- **public List<Trade> getTrades()** – returns the current list of **Trades** supported by this **Trader**
- **public void addRandomTrade()** – adds a new, randomly-chosen **Trade** to the list. The **Trade** should be generated with the following constraints:
    - The value for **gems** should be between 1 and 5 (inclusive)
    - The value for **amount** should be between 1 and 5 (inclusive)
    - The value for **Goods** should be randomly chosen from the values of the **Goods** enum

To generate a random number, you can use the **java.util.Random** class, as follows:

- Creating an object: **Random rand = new Random();**
- Later on in the code, when you need a number: **int value = rand.nextInt(n);** will return a number between 0 and (n-1), inclusive

Task 5: Trade.execute() (4 marks)

In your **Trade** class, implement one additional public method, as follows:

- **public void execute(Trader trader, Citizen citizen)**

This method should behave as follows:

- If the current **Trade** is not included in the list of trades supported by **trader**, this method should throw an **IllegalArgmentException**
- Otherwise, it should call **citizen.executeTrade()** with the current trade
    - If **executeTrade()** returns true, the method should also call **trader.addRandomTrade()**

Java submission

Ensure your submission files are named appropriately:

- Goods.java
- Trade.java
- Citizen.java
- Trader.java

Go to your moodle account, and submit your files at the appropriate submission link:

Moodle > Programming and System Design > Lab Exam > Java Submission