**Description**

Your task here is to implement a Java code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider default visibility of classes, data fields and methods unless mentioned otherwise.

**Specifications:**

**class** **definitions**:

**class** **ArrayListOps**:

 method definitions:

  convertArrayListtoInt(**int** n):

**return** type: ArrayList<Integer>

      visibilty: **public**

  reverse(ArrayList<Integer> list):

**return** type: ArrayList<Integer>

      visibilty: **public**

You don't need to implement the main() method. It has already been implemented as a part of the test-cases. It contains an **ArrayList** of integers.

**Task:**

Your task is to create a **Source**and implement the following:

* **convertArrayListtoInt(int n):**Method to create an **arrayList** with number of components **n** and ***set***components to **0**
* **reverse(ArrayList<Integer> list):** Method to ***Reverse* list**

**Important:**

* To check your program, you can use the **main()** method (in Source class) given in the stub. You can make suitable function calls and use **RUN CODE** button to check your main() function output.**﻿**

**Sample Input**

ArrayList<Integer> list = **new** ArrayList<Integer>(Arrays.asList(10, 25, 33, 28, 10, 12));

**Sample Output**

[0, 0, 0, 0]

[12, 10, 28, 33, 25, 10]

**Description**

You are tasked with implementing a Java program that models IP addresses and checks if they are valid IP addresses and whether they are multicast addresses. You need to create a base class called IPAddress with virtual member functions for these checks. Additionally, you are required to create two derived classes, IPv4Address and IPv6Address, which inherit from the IPAddress class and override the virtual functions according to the rules for IPv4 and IPv6 addresses.

**Class Definitions:**

-> **IPAddress**: This class represents an IP address.

* Attributes:
* ipAddress: represents the IP address.
* **Constructor**:
* Initialize the data members for an object of a class
* **isValid() (virtual function):**This function should return a boolean indicating whether the IP address is valid.
* **isMulticast() (virtual function):**This function should return a boolean indicating whether the IP address is a multicast address.

->**IPv4Address**: (create the class from existing class): This class represents an IPv4 address.

* **Constructor**:
* Initialize the data members for an object of a class
* **isValid():**Override this function to check if the IPv4 address is valid according to the IPv4 address format rules (e.g., "192.168.0.1"). Return true if valid, false otherwise.
* **isMulticast()**: Override this function to check if the IPv4 address is a multicast address. IPv4 multicast addresses have a specific range, usually starting with an address between 224.0.0.0 and 239.255.255.255. Return true if it's a multicast address, false otherwise.

->**IPv6Address** (create the class from existing class): This class represents an IPv6 address.

* **Constructor**:
* Initializes an IPv6 address object with the given string.
* **isValid():**Override this function to check if the IPv6 address is valid according to the IPv6 address format rules (e.g., "2001:0db8:85a3:0000:0000:8a2e:0370:7334"). Return true if valid, false otherwise.
* **isMulticast():**Override this function to check if the IPv6 address is a multicast address. IPv6 multicast addresses have a specific prefix, usually starting with "FF". Return true if it's a multicast address, false otherwise.

**Sample Input**

IPAddress ipv4Address1 = **new** IPv4Address("224.0.22.1");

ipv4Address1.isValid();

ipv4Address1.isMulticast();

**Sample Output**

true

true

**Description**

**Environment Specifications & Instructions**

Allowed Languages

* Java7
* Java8

**Build Expectation**

Problem Statement

Steve loves studying different data structures. Today, he studied about linked list and was fascinated by its features. He programmed it using *C* but has no clue how to implement it using **Java**(**Note**: He doesn't want to use inbuilt one.)

As you are famous in his town for Java skills, he came for your help.

He needs following things in his program.

**Specifications**

Class named Node: //For describing node of a linked list

Protected data members:

int value;

Node next;

Constructors:

Node(int num,Node n):

sets the **values** **of** **value** **and** **next** **data** members

**Public** methods:

void set\_next(Node n):

**Sets** the **value** **of** "next" **of** the **instance**

void set\_value(int d)

**Sets** the **value** **of** "value" **of** the **instance**

Node get\_next()

**Returns** the **value** **of** "next"

int get\_value()

**Returns** the **value** **of** "value"

Class named linkedlist: //for implementing the methods of linked list

Protected data members:

Node head;

Node tail;

Public data memeber:

int size;

Constructor :

linkedlist():

sets the value of head and tail equals to null and size equals to 0.

Public methods:

boolean Empty():

returns true if linkedlist is empty else false.

int get\_size():

returns the value of size.

public void **insert**(int v):

Creates a **new** **instance** **of** Node **class** **with** **value** "v" **and** **add** it **to** the linkedlist

**public** void **delete**(int **position**)

Deletes a node **from** the given **list** **from** the given **position**

**public** boolean is\_present(int **key**)

**returns** true **if** the "key" **is** **present** **in** the linkedlist **else** false.

**Task**

You just need to give the definition of Classes and it’s functions and rest of the needs will be taken care of.

**Description**

Your task here is to implement a **Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider default visibility of classes, data fields and methods unless mentioned otherwise.

**Specifications:**

**class** **definitions**:

**class** **Source**:

 method definitions:

  getDistinct(IntStream stream):Method to **return** the count of distinct element in the given stream

**return** type: **long**

      visibilty: **public**

**Task**

Create a**Source** class and implement below given method:

* **getDistinct(IntStream stream):**Method to return the **count** of distinct element in the stream

Implement using**Stream API**methods.

**Sample Input**

1, 3, 3, 5, 6, 6, 7

**Sample Output**

5

**Description**

**Problem Statement**

You are tasked with simulating a lift system that can move between different floors of a building and transport passengers. The system should maintain its movement operations and allow for user interaction through a command-line interface.

**Class Definitions:**

**Passenger:**

->Create a Java class named "**Passenger**" to represent passengers. Passengers should have the following attributes:

* **destinationFloor**: Represents the floor to which the passenger wants to go.
* Define the getter method for the attribute.

**Lift:**

->Create a Java class named "**Lift**" to represent the lift. The lift should have the following attributes:

* currentFloor : Represents the floor where the lift is currently located.
* state: The state of the elevator, which can be one of the following values:

"**stopped**": The elevator is not moving and the doors are closed.

"**moving** **up**": The elevator is moving upwards.

"**moving** **down**": The elevator is moving downwards.

* passengers: A list to keep track of passengers inside the lift.

->Initialize the Lift object with the following initial attributes:

* currentFloor is set to 1 by default.
* state is set to "stopped" by default.
* passengers is initialized as an empty ArrayList, representing that there are no passengers inside the elevator when it's created.

->Implement the following methods in the "Lift" class:

* getCurrentFloor(): Returns the current floor where the elevator is located.
* getState(): Returns the current state of the elevator (stopped, moving up, or moving down).
* getPassengers(): Returns a list of passengers currently inside the elevator.
* addPassenger(Passenger passenger): Adds a passenger to the lift.
* move(int destinationFloor):Moves the elevator to a specified destination floor. The elevator's state is updated based on the direction of movement. Returns true if the elevator successfully moves to the destination floor.
* openDoors(): Simulates the opening of the elevator doors on the current floor. Passengers whose destination is the current floor exit the elevator, and a list of exiting passengers is returned.
* closeDoors(): Simulates closing the lift doors after passengers have entered or exited.

**Sample Input**

Lift lift = **new** Lift();

lift.addPassenger(**new** Passenger(5));

lift.move(5);

lift.getCurrentFloor();

**List**<Passenger> exitedPassengers = lift.openDoors();

exitedPassengers.size();

**Sample Output**

true

5

1

**Description**

Your task here is to implement a Java code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider default visibility of classes, data fields and methods unless mentioned otherwise.

**Specifications:**

**class** **definitions**:

**class** **Source**:

      method definitons:

        sum(ArrayList<Integer> numbers):

**return** type: **int**

                            visibility: **public**

        index(ArrayList<Integer> list):

**return** type: ArrayList<Integer>

                            visibility: **public**

        retrieve(ArrayList<Integer> list):

**return** type: **int**

                            visibility: **public**

        main(String args[]): method of type **static** **void**

                        numbers:  ArrayList<Integer>

                        method calls:

                            sum(numbers)

                            index(numbers)

                            retrieve(numbers)

**Task:**

your task is to Create a **Source**class based on the above specifications and implement the following:

1. **sum (ArrayList<Integer> numbers)**method to add the elements in an ArrayList<Integer>.
2. **index (ArrayList<Integer> list)** method to modify the element in an ArrayList at index 2 and put the value 20.
3. **retrieve (ArrayList<Integer> list)** method to retrieve the element at index 1

**Description**

Design a Code to manage athletes and their participation in various Olympic events, track their medal counts, and calculate the total medals won by each athlete across all events.

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Athlete Class:**

**Properties:**

* name: The name of the athlete.
* country: The home country of the athlete.
* goldMedals: The number of gold medals won by the athlete.
* silverMedals: The number of silver medals won by the athlete.
* bronzeMedals: The number of bronze medals won by the athlete.

**Methods:**

* awardMedal(String medal): Award a medal to the athlete.
* getTotalMedals(): Return the total count of medals won by the athlete.
* getName(): Return the name of the athlete.
* getCountry(): Return the country of the athlete.

**Event Class:**

**Properties:**

* eventName: The name of the event.
* athletes(List of Athlete ) : A list of athletes participating in the event.

**Methods:**

* getEventName(): Return the eventName of the event.
* getAthletes(): Return the athletes of the event.
* addAthlete(Athlete athlete): Add an athlete to the event.
* getTopThreeAthletes(): Return the top three athletes based on their total medal counts.

**Olympics Class:**

**Properties:**

* events(List of Event ): A list of events in the Olympics.

**Methods:**

* addEvent(Event event): Add an event to the Olympics.
* addAthleteToEvent(Athlete athlete, Event event): Add an athlete to a specific event.
* getTotalMedalCount(): Calculate the total medal count for the top three athletes in each event.

Assumptions:

1. Each Athlete has a unique name and country.
2. Event has eventName, List of Athlete in athletes.
3. Olympics has name and List of Event in events.

**Sample Input**

Athlete athlete1 = **new** Athlete("Alice", "USA");

Athlete athlete2 = **new** Athlete("Bob", "Canada");

Athlete athlete3 = **new** Athlete("Charlie", "UK");

Event event1 = **new** Event("Swimming");

Event event2 = **new** Event("Running");

Olympics olympics = **new** Olympics();

olympics.addEvent(event1);

olympics.addEvent(event2);

olympics.addAthleteToEvent(athlete1, event1);

olympics.addAthleteToEvent(athlete2, event1);

olympics.addAthleteToEvent(athlete2, event2);

olympics.addAthleteToEvent(athlete3, event2);

athlete1.awardMedal("gold");

athlete2.awardMedal("silver");

athlete3.awardMedal("bronze");

athlete1.awardMedal("silver");

athlete2.awardMedal("gold");

athlete2.awardMedal("gold");

athlete3.awardMedal("silver");

olympics.getTotalMedalCount();

Sample Output

2

3

2﻿﻿﻿﻿

**Description**

Your task here is to implement a Java code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider default visibility of classes, data fields and methods unless mentioned otherwise.

**Specifications:**

class definitions:

  class Employee:

    data fields:

      name: String

      salary: int

    Implement a Constructor using the class variables.

    Implement getter setter methods **with** **public** visibility.

**class** EmployeeInfo:

    enum definition:

      named constants: BYNAME

                       BYSALARY

    method definitions:

**sort**(**List**<Employee> emps, **final** SortMethod method): Method **to** **return** sorted **list** **by** **name** **and** **by** salary **using** SortMethod

**Return** **type**: **List**<Employee>

           Visibility: **public**

      isCharacterPresentInAllNames(Collection<Employee> entities, **String** character): method **to** **check** **if** Employee **list** contains a **name** **starting** **with** a specific character

**Return** **type**: boolean

           Visibility: **public**

**Task:**

Create an Employee class which has the following members:

**String** name;

**int** salary;

* Define parameterized ***constructor***.
* Define ***getter*** method for all instance variables with public visibility.(getName(),...)
* Define ***setter*** methods for all instance variables with public visibility.(setName(),....)

Create an **EmployeeInfo** class which performs following operations (as per the given requirements) using **StreamAPI**:

* **enum SortMethod** : representing a group of named constants **BYNAME** and **BYSALARY**
* **sort(List<Employee> emps, final SortMethod method)**: Method to return sorted list by **name** and by **salary** using **SortMethod**
* **isCharacterPresentInAllNames(Collection<Employee> entities, String character):** Method to check if Employee list contains a name starting with a specific character

Implement using **Lambda expressions**.

Following has been done for you:

* **Main()** method containing list of **Employees**
* **String toString()** method, it's part of code stub, don't edit it else your***test-cases might fail***

**Sample Input**

List<Employee> emps = **new** ArrayList<>();

emps.**add**(**new** Employee("Mickey", 100000));

emps.**add**(**new** Employee("Timmy", 50000));

emps.**add**(**new** Employee("Annny", 40000));

**Sample Output**

[<name: Annny salary: 40000>, <name: Mickey salary: 100000>, <name: Timmy salary: 50000>]

[<name: Annny salary: 40000>, <name: Timmy salary: 50000>, <name: Mickey salary: 100000>]

false

**NOTE**

* Do not use any **for** loops or other control structures.
* Use the stream API methods for your implementations, else the test-cases might fail.
* You CAN implement the **main()** method to check the implementation of your methods in the solution.
* Upon implementation of **main()** method, you can use the **RUN CODE** button to pass input data in the method calls and arrive at some output.

**Description**

Your task here is to implement **Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider **default visibility** of classes, data fields, and methods unless mentioned.

**Specifications**

class definitions:

class Car﻿:

data member:

int carId

String carName

String type

visibility: private

﻿

﻿ Car(int carId, String carName, String type):constructor **with** **public** visibility

**Define** getter **and** setter **with** **public** visibility﻿

**class** Site:

**data** **member**:

int siteId

int carLimit

ArrayList<Car> cars

visility: **private**

Site(int siteId, **String** carLimit) : **constructor** **with** **public** visibility

﻿

method definition:

addCar(Car car):

**return** : **String**

visibility : **public**

getCarByType(**String** carType):

**return** : int

visibility : **public**

removeCarById(int **id**):

**return** : **String**

visibility : **public**

Task

**class Car**

- define data members according to the above specifications

-define a constructor and getters setters according to the above specifications

**class Site**

- define data members according to the above specifications

-define a constructor according to the above specifications

**-Implement the below methods for this class:**

**-String addCar(Car car):**

* ﻿Write a code to add a given car object to the cars Array list.
* The car will be added **if and only if the cars list has cars less than the carLimit variable.**
* If it is possible to add a car then return "**Car added!**" else return**"Site is full!".**

**-int getCarByType(String carType):**

* ﻿Write a code to count the number of cars in cars Array list with the same type as the given parameter carType.
* If the given carType is not **"Petrol", "Diesel" or "Electric" then return -1 else return the count of the cars.**

**-String removeCarById(int id):**

* ﻿Write a code to remove the car from the car's Array list if it is in the list.
* **If it is available and removed then return "Car out" else return "No car found"**

**Sample Input**

Car car1 = **new** Car(1001,"Mycar1","Petrol");

Car car2 = **new** Car(1002,"Mycar2","mytype");

Site site = **new** Site(12,30);

site.addCar(car1);

site.addCar(car2);

site.getCarByType("Petrol");

site.removeCarById(1002);

**Sample Output**

Car added!

Car added!

1

Car out

**Description**

Your task here is to implement a Java code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider default visibility of classes, data fields and methods unless mentioned otherwise.

**Specifications:**

class definitions:

  class Employee:

    data members:

      String name;

  int salary;

   Employee(String name, int string): constructor **with** **public** visibility

**define** getter **with** **public** visibility

**class** EmployeeInfo:

    method definitions:

      SortMethod:

**return** **type**: enum

          named Constants: BYNAME, BYSALARY

**sort**(**List**<Employee> emps, **final** SortMethod method):

**Return** **type**: **List**<Employee>

           Visibility: **public**

      getThirdInCollection(Collection<Employee> entities):

**Return** **type**: Employee

           Visibility: **public**

      isCharacterPresentInAllNames(Collection<Employee> entities, **String** character):

**Return** **type**: boolean

           Visibility: **public**

**Task:**

Create an **Employee** class which has the following members:

-String name;

-int salary;

* Define parameterized ***constructor***.
* Define ***getter*** method for all instance variables.(getName(),...)

Create an **EmployeeInfo** class which performs following operations (as per the given requirements) using **StreamAPI**:

* **enum SortMethod** : representing a group of named constants **BYNAME** and **BYSALARY**
* **sort(List<Employee> emps, final SortMethod method)**: Method to return sorted list by **name** and by **salary** using **SortMethod**
* **getThirdInCollection(Collection<Employee> entities)**: Method to return **third element**in list using **skip** method
* **isCharacterPresentInAllNames(Collection<Employee> entities, String character)**: Method to check if Employee list contains a**specific character** out

Implement using **Lambda expressions**.

Following has been done for you:

* **String toString()** method, it's part of code stub, don't edit it else your***test-cases might fail***

**Sample Input**

List<Employee> emps = **new** ArrayList<>();

emps.**add**(**new** Employee("Mickey", 100000));

emps.**add**(**new** Employee("Timmy", 50000));

emps.**add**(**new** Employee("Annny", 40000));

EmployeeInfo employeeInfo = **new** EmployeeInfo();

employeeInfo.sort(**new** ArrayList<>(emps), EmployeeInfo.SortMethod.BYNAME);

employeeInfo.sort(**new** ArrayList<>(emps), EmployeeInfo.SortMethod.BYSALARY);

employeeInfo.getThirdInCollection(emps);

String characterToCheck = "m";

employeeInfo.isCharacterPresentInAllNames(emps, characterToCheck);

**Sample Output**

**<name: Annny salary: 40000>**

**<name: Mickey salary: 100000>**

**<name: Timmy salary: 50000>**

**<name: Annny salary: 40000>**

**<name: Timmy salary: 50000>**

**<name: Mickey salary: 100000>**

**<name: Annny salary: 40000>**

**false**

**NOTE**

* Do not use any **for** loops or other control structures.
* Use the stream API methods for your implementations, else the test-cases might fail.
* You CAN implement the **main()** method to check the implementation of your methods in the solution.
* Upon implementation of **main()** method, you can use the **RUN CODE** button to pass input data in the method calls and arrive at some output.

**Description**

This is a short exercise in using **StreamApi**.

Your task here is to implement a **Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider default visibility of classes, data fields and methods unless mentioned otherwise.

**Specifications:**

**class** **definition**:

**class** **Source**:

 method definitons:

   countWords(String sentence, String letter):

**return** type: **long**

       visibilty: **public**

   isWholeNumber(String s):

**return** type: boolean

       visibilty: **public**

**Task:**

Create a**Source** class and implement below given method :

* **countWords(String sentence, String letter):** Method to **count** number of words, which begin from a specific letter.(Characters should all be in lowercase)
* **isWholeNumber(String s):** Method to check whether the *string* is a **whole** **number or Not.**

Implement using **Lambda expressions**.

**Sample Input**

Source source = **new** Source();

String sentence = "Example string with various words, some starting with specific letters.";

String letter = "s";

source.countWords(sentence, letter);

String testString1 = "123456";

source.isWholeNumber(testString1);

**Sample Output**

Number **of** words starting **with** 's': 4

123456 is whole number: true

**NOTE**

* Do not use any **for** loops or other control structures.
* Use the **stream API** methods for your implementations, else the ***test-cases might fail***.

**Description**

Your task here is to implement a Java code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider default visibility of classes, data fields and methods unless mentioned otherwise.

**Specifications:**

class definitions:

 class Person:

     data fields:

         private String name:

         private int age:

   method definitons:

       Person(String name, int age): define Constructor **with** **public** visibility.

**define** Getters **for** **name** **and** age.

**define** setters **for** **name** **and** age.

**class** StreamImplementation:

   method definitions:

       sumAge(**List**<Person> **list**):

**return** **type**: int

           visibilty: **public**

       printName(**List**<Person> **list**):

**return** **type**: **List**<**String**>

           visibilty: **public**

       printAge(**List**<Person> **list**):

**return** **type**: **List**<Integer>

           visibilty: **public**

**Task:**

Create a **Person**class which has the following private members:

* name String
* age int

Define parameterized constructor.

Define **getter** method for all instance variables (getName(),....)

Define**setter** methods for all instance variables (setName(),....)

Create a**StreamImplementation** class which performs operations (as per the given requirements) using **StreamApi**:

* Create a method **sumAge(List<Person> list)** to get the **sum of ages** above 50.
* Create a method **printName(List<Person> list)** that returns the **name**of all the persons.
* Create a method **printAge(List<Person> list)** that returns the**age** of all the persons.

Implement using **Lambda expressions**.

**Sample Input**

List<Person> users = **new** ArrayList<>();

users.**add**(**new** Person("Ann", 20));

users.**add**(**new** Person("Peter", 25));

users.**add**(**new** Person("Katty", 100));

users.**add**(**new** Person("Elly", 64));

StreamImplementation Impl = **new**  StreamImplementation();

Imp1.sumAge(users);

Imp1.printName(users);

Impl.printAge(users);

**Sample Output**

164

[Ann, Peter, Katty, Elly]

[20, 25, 100, 64]

**NOTE**

* Do not use any **for** loops or other control structures.
* Use the stream API methods for your implementations, else the test-cases might fail.
* You can implement the **main()** method to check the implementation of your methods in the solution.
* You can make suitable function calls and use **the RUN CODE** button to check your **main()** method output.

**Description**

Your task is to implement a **Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider default visibility of classes, data fields and methods unless mentioned otherwise.

Specifications

**class** **definition**:

**class** **Main**:

  method definiton:

getSum(String s):

**return** type: **int**

visibility: **public**

Task

Create a**Main** class and implement below given method:

* **getSum(String s):** read a line of integers and then return sum of all integers(Use StringTokenizer class)

Sample Input

2 5 7 9

Sample Output

23

**Description**

Your task is to implement a **JAVA** code based on the following specifications. Note that your code should match the specifications precisely. Unless mentioned otherwise, consider default visibility of classes, data fields, and methods. All the methods that you are implementing should be non-static.

**Specification:**

Class definitions:

  class StringPlay:

    data fields:

            int max

            boolean palindrome

            int convert

    StringPlay() constructor **with** **public** visibility.

**class** StringMethods:

method definitions:

    getMax(StringPlay sp, **String** **str**, char ch):

    visibility: **public**

**return** **type**:int

    checkPalindrome(StringPlay sp, **String** word):

    visibility: **public**

**return** **type**:boolean

    convertToInt(StringPlay sp, **String** val):

    visibility: **public**

**return** **type**:int

**Task:**

Class **StringPlay**

- define the **int** variable **max.**

**-**define the **boolean** variable **palindrome.**

- define the **int** variable **convert.**

**-**define a **constructor**  according to the above specifications.

Class **StringMethods**

* Implement the below methods for this class:

**-int** **getMax(StringPlay sp, String str, char ch)**: Return the total number of char ch present in string str and assign the value to sp variable max and return the same value.

* Example: str = "This is good" , ch = 'o' then resultant value = 2
* Example: str = "doselect Et le", ch='e' then resultant value = 3

**-boolean checkPalindrome(StringPlay sp, String word) :**Check if the String str is palindrome or not. Return true if yes else return false. Assign the result(true/false) to palindrome(sp object).

* Example: str = "This is good" then resultant is false.
* Example: str = "mom" then resultant is true.

**-int convertToInt(StringPlay sp, String val):**Convert the string str to int, return the int value and assign the value to suitable sp variable(convert). All the strings will contain only numbers.

* Example: str = "123" then resultant is 123.

**Sample Input**

**StringMethods** sm = new StringMethods();

**StringPlay** sp = new StringPlay();

**String** str="fgfgfgf";

**char** ch= 'g';

**String** word="mom";

**String** val = "123";

**Sample Output**

3

true

123

Your task here is to implement **Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider **default visibility** of classes, data fields, and methods unless mentioned.

**Specifications**

class definitions:

class Schema:

data member:

int id

String name

float marks

char grade

visibility: private

﻿﻿ Schema(int id, String name, float marks, char grade): constructor **with** **public** visibility

Implement getter **and** setter methods **with** **public** visibility

**class** **Validation**:

**data** **member**:

**Schema** **schema**

﻿

**Validation** () : **Constructor** **with** **public** visibility

method definition:

**validate**(**String** **Data**):

**return** : **String**

visibility : **public**

﻿

printData():

**return** : **String**

visibility : **public**﻿﻿

Task

**class Schema**

- define this class according to the above specifications

**class Validation**

- define data members according to the above specifications

-define a constructor to the above specifications

-Implement the below methods for this class:

**-String validate(String Data):**

* ﻿Write a code to validate the given data according to the schema.
* The data provided will be in String format **separated by a comma.** Each comma-separated data **describes the schema fields.**

1. The first element denotes id.
2. The second element denotes name.
3. The third element denotes marks.
4. The fourth term denoted grade.

* Return "**Invalid data**" in the following conditions.

1. If any of the mentioned data cannot be converted to its respected without any exception.
2. If the number of elements in the provided string is more than 4.
3. If the data type of all the above-mentioned elements is not the same.
4. If the grade is other than '**A', 'B' and 'C'.**

* If the given data satisfy all the details then create a new Schema object and assign it to the class attribute schema and return "**Valid data**".

**-String printData():**

* If no value is assigned to the attribute schema than return **"Error**" else return the String in the following format - "**{name} has got {marks} with grade {grade}**". where the text in the curly braces( {} ) denotes the data of the object schema.

*Data = "1,MyName,12.7,A"*

*validate method will return true*

*Data = "2,MyName,12.1,Y"*

*validate method will return false*

**Sample Input**

Validation validation = **new** Validation();

validation.validate("1,MyName,12.5,A")

**Sample Output**

**Valid** data

**Explanation**

The passed string has 4 elements separated by a comma and the

1. The first element is an integer(1) that is id.
2. The second element is String(MyName) which is the name.
3. The third element is Float(12.5) that is marked.
4. The fourth element is Character(A) that is grade.

Therefore the returned String is "Valid data".

**Description**

Your task here is to implement **Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider **default visibility** of classes, data fields, and methods unless mentioned.

**Specifications**

class definitions:

class Customer:

data member:

String name

long mobile

﻿ char key

visibility: public

﻿﻿ Customer(String name, long mobile, char key):constructor with public visibility

method definition:

﻿encrypt():

return : String﻿﻿

visibility : public

minCount():

return : long

﻿visibility : public

Task

**class Customer**

- define data members according to the above specifications

-define a constructor to the above specifications

-Implement the below methods for this class:

**-String encrypt():**

* ﻿Write a code to calculate encrypt the name of the customer.
* If the name already contains that key at any index independent of occurrence then returns "**Already Encrypted".**
* If the name does not contain a key at any index independent of occurrence then return change the name instance with the encrypted name and also return the encrypted name with the following rule -

1. It gets the **first digit (x) of the mobile number.**
2. Then it **inserts the key x times at the last of the name.**

**-long minCount():**

* ﻿Write a code that returns the minimum digit of the mobile number. For example - the minimum digit of the mobile number **"98979593**" is **3.**

**Sample Input**

Customer customer = **new** Customer("Doselect", 299999992, '#');

System.**out**.println(customer.encrypt());

System.**out**.println(customer.minCount());

**Sample Output**

Doselect##

﻿2

**Explanation**

The minimum digit of the mobile number (99999992) is 2; Therefore two "#" are inserted in the name.

**Note -**The output string can be different as the key has to be inserted at different indexes.

**Description**

Given an array of **distinct** integers **arr**, your task here is to find all pairs of elements with the minimum absolute difference of any two elements.

Your task is to implement a **Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider default visibility of classes, data fields and methods unless mentioned otherwise.

**Specifications**

**class** **definitions**:**class** **Main**:

method definition:

minimumDifference(**int**[] arr):

**return** type: List<List<Integer>>

visibility: **public**

**Task:**

Create a class**Main**and implement the below given method:

* **List<List<Integer>>** **minimumDifference(int[] arr) :** Return a list of pairs in ascending order(with respect to pairs), each pair [a, b] follows

1. a, b are from arr
2. a < b
3. b - a equals to the minimum absolute difference of any two elements in arr

**Sample Input1**

arr = {4,2,1,3}

**Sample Output1**

[[1,2],[2,3],[3,4]]

Sample Input2

arr = {12, 2, 5, 9, 11, 22, 25}

Sample Output2

[[11, 12]]

**Explanation**

**For Input 1:**The minimum absolute difference is 1. List all pairs with difference equal to 1 in ascending order.

**NOTE**

* The above **Sample Input** and **Sample Output** are only for demonstration purposes and will be obtained if you implement the **main()** method with all method calls accordingly.
* Upon implementation of **main()** method, you can use the **RUN CODE** button to pass the **Sample Input** as input data in the method calls and arrive at the **Sample Output**.

**Description**

Your task here is to implement a Java code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider **default visibility** of class unless mentioned otherwise.

**Specifications:**

**class** **definitions**:

**class** **Age**:

**class** **variables**:

            String drink

            String vote

            String movie

**class** **Exceptionlegal**" **and** **throw** **an** **user**-**defined** **exception** **IllegalAgeException**("**Illegal** **movie**-**watching** **age**") **which** **extends** **the** **Exception** **class**.

* If the age is greater than 14, assign movie variable as "**legal**".
* Return default message if an exception is thrown else return **a.movie**.

**Important:**

* To check your program you have to use the **main()** function (in Source class) given in the stub. You can make suitable function calls and use **RUN CODE** button to check your **main()** function output.

**Description**

Your task here is to implement **Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider **default visibility** of classes, data fields, and methods unless mentioned.

**Specifications**

**class** definitions:

**class** Racer:

data member:

String action

**int** runScore

**int** jumpScore

**int** crawlScore

visibility: public

﻿

﻿﻿ Racer(String action, **int** runScore, **int** jumpScore, **int** crawlScore):constructor with public visibility

method definition:

﻿ goodAt():

**return** : String

﻿ visibility : public

﻿finalScore():

**return** : **int**

﻿ visibility : public﻿

Task

**class Racer**

- define **data members** according to the above specifications

-define a constructor according to the above specifications

-The term/variable used are defined below -

1. **action** - A string that contains only 3 different characters **'r', 'j', and 'c'**. Where **'r' - run, 'c' - crawl, and 'j' - jump.**
2. **runScore** - A int that denotes what score will he get for every 'r' in **action** string.
3. **jumpScore** - A int that denotes what score will he get for every 'j' in **action** string.
4. **crawlScore** - A int that denotes what score will he get for every 'c' in **action** string.

**-Implement the below methods for this class:**

**-String goodAt():**﻿

* Write a code that returns the string on the basis of the given conditions -
* If a count of the character 'j' is equal to the count of the character 'c' in the string **action**then return **"Perfect".**
* If a count of the character 'j' is greater than the count of the character 'c' in the string **action**then return **"Jumper".**
* If a count of the character 'j' is less than the count of the character 'c' in the string **action**then return **"Crawler".**

**-int finalScore():**

* Return the sum of the score for each action.

1. For every **'r'** in action, the score will be **runScore.**
2. For every**'j'** in action, the score will be**jumpScore.**
3. For every**'c'**in action, the score will be **crawlScore.**

* Refer Example for better understanding

***action = "jjcr" , runScore = 20, jumpScore = 30, crawlScore = 5***

***action.goodAt()****returns****"Jumper"****as****'j' count > 'c' count****in****action.***

***action.finalScore()****returns****85****as****20+20+5+30 = 85.***

**Sample Input**

Racer racer = **new** Racer("jjccrrj",10,20,30);

racer.goodAt();

**Sample Output**

**Jumper**

**Description**

Your task here is to implement **Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider **default visibility** of classes, data fields, and methods unless mentioned.

**Specifications**

class definitions:

class Knight:

data members:

﻿String name

List<String> skills

int power

visibility:private

﻿﻿ Knight(String name, List<String> skills, int power):

﻿constructor with public visibility

Implement getter setters with public visibility

class Moon:

﻿data members:

ArrayList<Knight> knights

visibility : public

﻿ method definition:

trainKnight(Knight knight):

return : String

visibility: public

﻿

﻿searchBy(String query):

return : int

visibility : public

**class Knight**

- define data members according to the above specifications

-The skills variable contains a list of skills separated by a comma.

-Assuming that all the strings are in lowercase.

-define a constructor according to the above specifications

**class Moon**

- define data members according to the above specifications

-Implement the below methods for this class:

**-String trainKnight(Knight knight):**

* Write a code to add a knight to the given list for the following validations.
* The power of the knight must be greater than 75 if not return "**Less power".**
* If there are less than 3 skills present then return "**Train harder"**
* If a knight of the same name is present in the given list then return "**Already present".**
* **If all these conditions get fulfilled return "Welcome to the training" after adding the given object to the given list.**

**-int searchBy(String query):**

* The query variable will have two parts separated by '-'.
* If the first part of the query variable is equal to "**power**", then check if each **power** variable in the **Knight** list matches the second variable of the query. If the condition is true, return the count of the **power**, i.e, you have to do the power count.
* If the first part of the query variable is not equal to "**power**", then check if each **name** variable in the **Knight** list matches the second variable of the query. If the condition is true, return the count of the **name**, i.e, you have to do the name count.
* If no match is found return 0.

**Sample Input**

List<String> skills = **new** ArrayList<>();

skills.**add**("fly");

skills.**add**("run");

skills.**add**("super-strength");

skills.**add**("armor");

Knight kn1=**new** Knight("Steven",skills,80);

Moon mn=**new** Moon();

mn.trainKnight(kn1);

**Sample Output**

**Welcome** to the training

**Description**

**Problem Statement**

Imagine you are tasked with developing a program to manage a home electrical supply system that comprises multiple rooms and various electrical devices. Your goal is to create a system that can turn devices on and off, as well as measure the total power consumption within each room.

**ElectricalDevice Class:**

The ElectricalDevice class represents a generic electrical device and has the following data members:

* **deviceName**: Represents the name of the device installed in a room.
* **powerConsumption**: Represents the power consumption (in watts) of the device.
* **isOn**: Represents whether the device is turned on or off.

You should assign values to these data elements when an ElectricalDevice object is initialized.

Implement the following methods within the ElectricalDevice class:

* **getPowerConsumption**: This method should return the power consumption of the device, taking into account whether it's turned on or off.
* **turnOn:**This method is used to turn the device on.
* **turnOff**: This method is used to turn the device off.
* **isOn**: This method should return the status of the device (i.e., whether it is on or not).
* **isOff**: This method should return the status of the device (i.e., whether it is off or not).

**Room Class:**

The Room class represents a residential room, which can be a living room, bedroom, guest room, or any other type of room in a house.

Add the following data member to the Room class:

* **devices**: Represent a list of devices installed in the room, such as fans, lamps, laptops, Wi-Fi routers, and more.
* **name**: Represents the name of the room, such as "Living Room," "Bedroom," or "Guest Room."

You should assign values to the Name data element when a Room object is initialized.

Implement the following method within the Room class:

* **addDevice**: Add an electrical device to the room.
* **getTotalPowerConsumption:** Calculate and return the total power consumption of all the active devices in the room.

*Total Power Consumption = Σ (Power Consumption of Each Active Device)*

**Sample Input**

ElectricalDevice laptop = **new** ElectricalDevice("Laptop", 50.0);

laptop.turnOn();

**double** laptopPowerConsumption = laptop.getPowerConsumption();

laptop.getPowerConsumption();

ElectricalDevice refrigerator = **new** ElectricalDevice("Refrigerator", 150.0);

refrigerator.turnOn();

**double** refrigeratorPowerConsumption = refrigerator.getPowerConsumption();

refrigerator.getPowerConsumption();

Room guestRoom = **new** Room("Guest Room");

guestRoom.addDevice(laptop);

guestRoom.addDevice(refrigerator);

**double** totalPowerConsumption = guestRoom.getTotalPowerConsumption();

guestRoom.getTotalPowerConsumption();

**Sample Output**

50.0

150.0

200

**Description**

**Problem Statement**

You are tasked with designing a program to manage orders for a coffee shop. The program should be able to handle different types and sizes of coffee and calculate the total cost for each order. Below is the design for this problem:

Classes:

**Coffee:**

->Properties:

* type: The type of coffee (e.g., "Regular," "Latte," "Cappuccino," "Espresso").
* size: The size of the coffee (e.g., "Small," "Medium," "Large").
* quantity: The quantity of this coffee in the order.
* Assign values to data elements when the object is initialized.

**Order:**

->Properties:

* customerName: The name of the customer placing the order.
* coffeeList: A list of coffee items in the order.
* Initialize the order with the customer's name and the list of coffee items.

**CoffeeShop:**

->Constants:

* PRICES: Stores the prices of different coffee types and sizes.

*{2.00, 2.50, 3.00}, // Regular*

*{3.00, 3.50, 4.00}, // Latte*

*{3.50, 4.00, 4.50}, // Cappuccino*

*{2.50, 3.00, 3.50}  // Espresso*

->Methods:

* calculateTotalCost(Order order): Calculate and return the total cost of the order based on the coffee items in the order. Use the PRICES array to look up the prices of each coffee type and size.

**Sample Input**

List<Coffee> coffee = **new** ArrayList<>();

coffee.**add**(**new** Coffee("Regular", "Small", 27));

coffee.**add**(**new** Coffee("Latte", "Medium", 5));

coffee.**add**(**new** Coffee("Cappuccino", "Large", 3));

List<Coffee> coffee1 = **new** ArrayList<>();

coffee1.**add**(**new** Coffee("Espresso", "Small", 2));

coffee1.**add**(**new** Coffee("Latte", "Large", 1));

Order order = **new** Order("customer 1", coffee);

Order order1 = **new** Order("Customer 2", coffee1);

CoffeeShop.calculateTotalCost(order));

CoffeeShop.calculateTotalCost(order1));

**Sample Output**

85.0

9.0

**Description**

Your task here is to implement a**Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider **default visibility** of classes, data fields and methods are public unless mentioned otherwise.

**Specifications**

class definitions:

﻿class Product:

Data members:

﻿﻿ String name;

﻿ double price;

﻿ String coupon;

public Product(String name,double price,String coupon): constructor **with** **public** visibility.

**class** Validator:

method definitions:

﻿ validateCoupon(Product p)throws **Exception**

﻿**return** **type**: **String**

visibility: **public**

netPrice(Product p)

**type**: **double**

visibility: **public**

**class** InvalidCouponException:

﻿ method definition:

﻿ InvalidCouponException(**String** msg)

visibility: **public**

**Task**

Class **Product**

**-**define the **String** variable **name**

**-**define the **double** variable **price**

**-**define the **String** variable **coupon**

-define the constructor as per given in the specifications.

Class **Validator**

Implement the below methods for this class:

-**String** **validateCoupon(Product p):**

* throw an **InvalidCouponException** "Invalid Coupon" if the coupon is not valid. The **coupon is valid**if its name and discount value are separated with '-' and the discount value should be between 10-25(inclusive).

**Example:**

**name**= "**IPhone**" ; valid **coupons** are "**IPhone-10**", "**IPhone-20**", "**IPhone-18**" etc.

* **return** "Valid Coupon" if no exception found.

-**double netPrice(Product p):**

* **netPrice** = totalPrice-discountPrice.
* **return** netPrice if Coupon is valid else **return** totalPrice.

Class **InvalidCouponException**

* define custom exception class **InvalidCouponException** by **extending** the **Exception** class.
* define a parameterised constructor with a String argument to pass the message to the super class.

**Sample Input**

Product obj = **new** Product("IPhone",25000,"IPhone-10");

Validator val = **new** Validator();

val.validCoupon(obj);

val.netPrice(obj);

**Sample Output**

valCop = "Valid Coupon"

price = 22500.0

**Description**

You are given a string, s1, containing a list of IP addresses and potentially some invalid inputs separated by semicolons.

Design the below class.

**class Classification**

Your task is to implement a function, find, which validates and extracts valid IP addresses from the input string, given a positive integer, num, representing the expected number of valid IP addresses.

*A valid IP address is defined as a string consisting of four non-negative integers, each between 0 and 255 (inclusive), separated by periods ('.'), and having exactly three periods.*

The function find should take the following parameters:

* num representing the expected number of valid IP addresses (1 <= num <= 100).
* s1 containing a list of IP addresses separated by semicolons (';') and potentially some invalid inputs. The length of s1 is at most 1000 characters.

The function should return a list of strings containing the valid IP addresses from the input string s1. If the input does not contain the expected number of valid IP addresses (i.e., less than num valid IP addresses), or if num is less than 1, the function should return a list containing a single string: "Invalid Input".

Sample Input

int num = 3;

String s1 = "192.168.1.2;192.168.1.7;192.168.1.10;";

List<String> result = find(num, s1);

Sample Output

 192.168.1.2

 192.168.1.7

 192.168.1.10

**Description**

**Problem Description**

You are tasked with designing a **Java** program to manage water supply within a society. The society consists of individual flats, each with the ability to consume water. The goal is to ensure that water supply is synchronized and consumed in an orderly manner.

**WaterSupplyManagement Class:**

The WaterSupplyManagement class represents the society's water supply management system. It tracks the available water and allows for consumption. When an object of this class is initialized, the initial water tank capacity is assigned.

Data Members:

* availableWater: represents the capacity of water in the society's tank.

->Create the instance of the class

Methods:

1. **consumeWater**: A synchronized method used when a flat needs to consume water. Deduct the consumed water from the available water and return the amount consumed along with the updated available water.
2. **getAvailableWater**: Return the current available water.

**Flat Class:**

The Flat class represents individual flats in society. Each flat can consume water and stop using water.

Data Members:

* **waterSupplyManagement**: An instance of the WaterSupplyManagement class.
* **flatNumber**: Represents the flat number.

->Create the instance of the class

Methods:

1. **useWater**: A method to consume water for the flat. Call the consumeWater method of the WaterSupplyManagement class and return the amount of water consumed.
2. **stopUsingWater**: A method to stop water consumption by the flat. Return a message indicating that the flat has stopped using water.

Sample Input:

WaterSupplyManagement society = **new** WaterSupplyManagement(10000);

Flat flat101 = **new** Flat(101, society);

flat101.useWater(2000);

Flat flat102 = **new** Flat(102, society);

flat102.useWater(2500);

flat101.stopUsingWater();

Flat flat103 = **new** Flat(103, society);

flat103.useWater(1500);

flat102.stopUsingWater();

Flat flat104 = **new** Flat(104, society);

flat104.useWater(3500);

Sample Output:

8000

5500

Flat 101 stopped **using** water.

4000

Flat 102 stopped **using** water.﻿

500

Note:

* The consumeWater method is synchronized to ensure that only one flat can consume water at a time.
* The output shows the amount of water consumed by each flat and the updated available water after each consumption.

**Description**

Your task here is to implement **Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider **default visibility** of classes, data fields, and methods unless mentioned.

**Specifications**

class definitions:

class Customer:

data members:

String customerName

int amount

﻿ int time

int creditScore

visibility: public

﻿﻿ Customer(String customerName, int creditScore,int amount,int time): constructor

﻿

class Bank

data members:

List<Customers> bankCustomers

List<Customers> merchantCustomers

﻿ int rate = 10

method definitions:﻿﻿

addCustomer(Customer customer):

return: String

visibility public

﻿ subtractAmount(String data):

return : int

visibility : public﻿

**class Customer**

- define data members according to the above specifications

-define a constructor and initialize the values.

**class Bank**

- define data members according to the above specifications

-initialize the rate variable with 10.

-Implement the below methods for this class:

**-String addCustomer(Customer customer):**

* ﻿Write a code to add a given customer object to the provided ArrayList after verification.
* If the creditScore of the customer is less than (amount\*100\*time) then add the given customer into merchantCustomer ArrayList and return "Added to Bank".
* Else add into bankCustomer ArrayList then return "Added to Merchant". ﻿

**-int** **subtractAmount**(**String data**):

* ﻿Write a code that calculates the amount after installment.
* Given the string data that contains the installment amount, total loan amount, and paid amount.
* If the installment amount is greater than paying amount then the bank lays an extra charge at the bank rate on the amount gap(installment amount-paying amount) for one year. Subtract the paying amount from the total amount loan and add the extra charges into the total loan amount.
* Else the bank charges no charge and substracts the paying amount from the total amount loan.

**Sample Input**

Bank bank=**new** Bank();

Customer cst=**new** Customer("A",2000,50000,2);

bank.addCustomer(cst);

**Sample Output**

**Added** to Bank

**Description**

**Problem Statement**

A portal is to be created for a hotel to manage room bookings for their customers and provide information on room availability and bookings. The system consists of several classes, each with its own set of properties and methods:

**1. Room**

-> Represent a hotel room.

Properties:

* **roomNumber** - Represents the room number.
* **roomType** - Represents the room type (e.g., STANDARD, DELUXE, SUITE).
* **roomStatus** - Represents the room status (e.g., AVAILABLE, BOOKED).
* **bookingPrice** - Represents the room booking price.
* **startDate** - Represents the check-in date.
* **endDate** - Represents the check-out date.

-> For instance variable roomNumber, return its value.

->Initialize the data members on object creation.

Method:

* **isRoomAvailable**- Return true if the room is available for the specified check-in and check-out dates; otherwise, return false.

**2. RoomBooking**

->Manages the bookings for a room.

Properties:

* **reservationNumber** - Represents the reservation number.
* **startDate** - Represents the check-in date.
* **endDate** - Represents the check-out date.
* **durationinDays** - Represents the stay duration.
* **status** - Represents the booking status (e.g., CONFIRMED, CANCELLED, PENDING).
* **advancePayment** - Represents the advanced payment amount.
* **roomList** - Represents the list of rooms booked as part of this booking.
* **dataLoader** - Represents the class used to store rooms and room booking data.

->Initialize dataLoader attribute on object creation.

Methods:

* **createRoomBooking** - Create a booking for the given reservation number and update the room status to BOOKED.
* **fetchRoomBookings** - Fetch room booking details for the given reservation number.

**3. RoomSearch**

->Responsible for searching available and booked rooms.

Properties:

* **dataLoader** - Used to access room and room booking data.

->Initialize the data member on object creation.

Methods:

* **searchAvailableRooms** - Search and return a list of available rooms matching the specified room type and dates.
* **searchBookedRooms**- Search and return a list of booked rooms matching the specified room type and dates.

**4. DataLoader**

->Singleton class used to store rooms and room booking data in collections.

Properties:

* **roomsMap** - Stores room numbers as keys and Room objects as values.
* **roomBookingMap** - Stores booking IDs as keys and RoomBooking objects as values.

-> Initialize the data members on object creation and should only be visible within the class, not from any other class (including subclasses).

Methods:

* **getInstance** - Return an instance of the DataLoader class (as it's a singleton class).
* **initialize** - Populate roomsMap with the given list of rooms.
* **createRoomBookings**- Update room status to BOOKED, create a RoomBooking object and store it in roomBookingMap.
* **getRoomBookings** - Return a RoomBooking from roomBookingMap based on the given reservation number.

**Enumerations**:

* **BookingStatus** - Values: CONFIRMED, CANCELLED, PENDING
* **RoomType** - Values: STANDARD, DELUXE, SUITE
* **RoomStatus** - Values: AVAILABLE, BOOKED

**Sample Input:**

Date checkInDate = **new** Date(2023, 1, 1);

Date checkOutDate = **new** Date(2023, 1, 2);

List<Room> rooms = **new** ArrayList<>();

rooms.**add**(**new** Room("F1", RoomType.STANDARD, RoomStatus.BOOKED, checkInDate, checkOutDate));

rooms.**add**(**new** Room("F2", RoomType.STANDARD, RoomStatus.AVAILABLE, checkInDate, checkOutDate));

rooms.**add**(**new** Room("F3", RoomType.STANDARD, RoomStatus.AVAILABLE, checkInDate, checkOutDate));

DataLoader dataLoader = DataLoader.getInstance();

dataLoader.initialize(rooms);

RoomBooking roomBooking = **new** RoomBooking(dataLoader);

RoomBooking createdBooking = roomBooking.createRoomBooking("RB1", checkInDate, checkOutDate, rooms);

RoomSearch roomSearch = **new** RoomSearch(dataLoader);

List<Room> availableRooms = roomSearch.searchAvailableRooms(RoomType.STANDARD, checkInDate, checkOutDate);

availableRooms.**get**(0).getRoomNumber();

availableRooms.**get**(1).getRoomNumber();

**Sample Output:**

**F2**

F3

**Description**

Your task here is to implement a Java code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider **default visibility** of class unless mentioned otherwise.

class definitions:

    Employee:

        class Variables:

            private String firstName

            private String lastName

            private String ssn

        Constructor:

            Employee():Empty constructor to intialise the instance variable as null.A testcase will **check** **for** the **creation** **of** **empty** constructor.

                visibility: **public**

            Employee(firstName,lastName,ssn): arameterized **constructor** **to** initialize the **instance** variables.

                visibility: **public**

        Getter Methods:

            getFirstName(): **Return** the firstName.

                Visibility: **public**

**Return** **type**: **String**

            getLastName(): **Return** the lastName.

                Visibility: **public**

**Return** **type**: **String**

            getSsn(): **Return** the ssn.

                Visibility: **public**

**Return** **type**: StringValidation Methods:

            validateName(**String** firstName, **String** lastName): Implement this **function** **with** three **Exception**- Explained below **in** Task section.

**Use** try-catch **block** **to** implement **exception** **and** **return** the suitable **Exception** Messages **from** the catch block.

**If** the firstName **and** lastName **is** valid **then** assign the firstName, lastName **to** appropriate **Class** **variable** **and** **return** "Valid String".

                Visibility: **public**

**Return** **type**: **String**

            validateSsn(**String** ssn): **Check** **if** the **first** **and** **last** character **of** the ssn **is** digit **return** "Valid String" **else** **return** "Invalid String"

                Visibility: **public**

**Return**-**type**: **String**

**Your Task is to:**

* Implement the **Employee**class according to the above specification.
* Employee class has three private variable : **firstName, lastName, ssn.**
* **Employee**Class contains three getter method. Implement the getter methods first and then implement the validation methods. Strictly follow the above specification order.

**Validation Methods:**

**1. validateName(String firstName, String lastName):**The three exception to be checked are:

* First if the firstName or lastName is **null**throw NullPointerException("Entry Missing").
* Second if the firstName or lastName length is zero throw StringIndexOutOfBoundsException("Index out of bound").
* Third if the firstName or lastName starts with a number throw IllegalArgumentException("First Character is Invalid").
* If the firstName and lastName is valid then assign the firstName, lastName to appropriate Employee Class variable and return "Valid String".

Use **try block** to check for the three exceptions and use **catch block** to return the suitable exception message(For each exception, messages are given in the specification eg. for **StringIndexOutOfBoundsException ("Index out of bound")**return message should be "**Index out of bound**" ). Both this function has a **String** return type.

**2. validateSsn (String ssn)** : Check if the first and last character of the ssn is digit(0-9) return "Valid String" else return "Invalid String".

**Description**

Your task here is to implement a**Java** code based on the following specifications. Note that your code should match the specifications in a precise manner.  Consider**default visibility** of classes, data fields and methods unless mentioned otherwise.

**Specifications:**

class definitions:

class Email:

Variables:

Header header

String body

String greetings

Implement a parameterized constructor to initialize all the instance variables.

class Header:

Variables:

String from

String to

Implement a parameterized constructor to initialize all the instance variables.

class EmailOperations:

Methods:

emailVerify(Email e): **Use** regular expression **to** **verify** **if** the two email-ids **in** the Header **class** **is** valid **or** not.[**Return** **type** explained **in** Task part].

**Return** **type**:int

Visibility: **public**

bodyEncryption(Email e): **Use** Ceasar cipher(Shift-3) **to** **encrypt** the **body** **of** the email.[**To** know more refer the Task part]

**Return** **type**:**String**

Visibility: **public**

greetingMessage(Email e): **In** this method you have **to** **return** a greeting messgae. The greet part should be taken **from** greetings **variable** **and** signature(**name**) should be taken **from** Header's '**from**' email address.[To know more refer the Task part]

Return type:String

Visibility: public

**Class Variables:**

* **class Header:**It contains two email id 'from' and 'to'. 'from' signifies the sender's email address and 'to' signifies receiver's email address.
* **class Email:**This class contains three parts: first **Header header** which has two email address **from** and **to**,the second **body** which contains the message to send and third **greetings** which contains greetings such as "Regards", "Thank you", etc.

To access a variable in Header class through Email object we use:

<**Email(obj)**>.<**Email(variable)**>.<**Header(variable)**>

Example to access "**from"**address from the Email object e we use : e.header.from;

**Tasks:**

* Implement the two classes **Email** and **Header** class according to the specifications.
* Implement the three methods in the **EmailOperations** class:

1. **emailVerify (Email e)**
2. **bodyEncryption (Email e)**
3. **greetingMessage (Email e)**

**Method Description:**

**1. emailVerify(Email e):**

* In this method you have to use regex to check if the email-address**to**and **from** in Header class is valid or not.Validation is based on:
* Email address should start with alphabets(capital/small) or \_(underscore).
* Email address should have only one @.
* Email address should end with .(dot) followed by alphabets.
* **e.g**: amit@doselect.com, \_ami@doselect.in are valid addresses, but 1ami@dos.com, amit@doselect are invalid addresses.
* Return 2 if the both email addresses are valid return 1 if one is valid, and 0 if both are invalid.

**2. bodyEncryption(Email e):**

* In this method, you have to use Caesar cipher(shift of 3) to encrypt the **body**part of the **Email** return the encrypted string.
* **Caesar shift**, is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. Here the number of shift is 3.
* **e.g**: str = "**Hi There Hows you**", after encryption becomes "**Kl Wkhuh Krzv brx**". H get converted to K that is a shift of 3 alphabets ahead.
* Letters which are capital should be capital and small should be small in Encrypted message. Take care of the spaces.

**3. greetingMessage(Email e):**

* In this method, you have to return a concatenated string which contains the greetings variable from Email class and Name of the person who is sending the mail(from variable in the Header class).
* The name part should not contain anything which is after @ in the email id.
* **e.g**: if greetings = "Regards" and from = "Amit@doselect.com" then you have to return the message "Regards Amit"

**Description**

Garry has recently learned about strings in her programming classes. She decided to create some interesting strings using the concepts. She finds it interesting to toggle the alphabet case in the string.

Help Garry understand the operations.

**﻿**Your task here is to implement a**Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider **default visibility** of classes, data fields and methods are public unless mentioned otherwise.

**Specifications**

**class** **definitions**:

﻿ **class** **Toggle**:

toggleString(String string):

**return** type: String

**Task:**

**class Toggle**

**Implement the below method for this class**:

* **String toggleString(String string):**Given a string,  toggle the alphabet case and return the string

**Sample Input1**

**abcd**

**Sample Output1**

**ABCD**

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class Validator:

method definitions:

checkComment(String **comment**) throws **Exception**:

**return** **type**: **String**

visibility: **public**

﻿

commentTheString(**String** **comment**) throws **Exception**:

**return** **type**: **String**

visibility: **public**

﻿

**class** SpamCommentException:

method definitions:

SpamCommentException(**String** msg)

visibility: **public** ﻿

**Task**

Class **Validator**

**Implement the below methods for this class:**

**-String** **checkComment(String comment):**

* Write a code to validate the comment.
* throw a **SpamCommentException,** if comment has these words["**abcde**", "**lmno**", "**pqrst**", "**wxyz**"] in it, with the message "**spam comment**".
* throw a **SpamCommentException**, if a comment contains more than 2 words from the above list, with the message "**account ban due to spam comment**". Note same words with a frequency of more than 2 will come in this category.
* return a string message "**comment is not spam**" If none of the above exceptions is found.

*Refer the example for better understanding.*

*s0 = "hello my name is steve and using****abcde****"*

*s1 = "hello my name is steve and using****abcde abcde****"*

*s2 = "hello my name is steve and using****abcde lmno pqrst****"*

*s3 = "hello my name is steve and using****abcde abcde lmno****"*

*s0 and s1 come under****spam comment****message.*

*s2 and s3 comes under the****account ban****message.*

**-String commentTheString(String comment):**

* Write a code to put the comment on the post.
* If a checkComment method throws a **SpamCommentException** ,then return a message of that exception(Use try-catch block).
* If it throws any other exception then return a message "**other exception**".
* If no exception is found then return a message "**comment posted**".

class **SpamCommentException**

-Define **SpamCommentException** class derived from Exception class

**Sample Input**

Validator obj = new Validator();

-------------------------------------------

obj.checkComment("hello my name is steve");

obj.commentTheString("my **comment** **is** **safe** **to** post");

**Sample Output**

**comment** **is** **not** spam

**comment** posted

**Description**

Danish has opened a seed bags selling shop. He wants to arrange the bags in the order in which a bag expires first. These bags have a twelve-digit code where:

* the first four characters are Batch Number.
* The next eight digits represent the seed expiry date in **YYYYMMDD** format.

Write a Java code to extract the expiry date from package code and display the same along with the Batch Number.

**Validations:**

* Batch number is valid only if the first, second and fourth characters are **letters** of the alphabet (UpperCase) and the third character is a **number**.
* If the year is not between 2015 and 2020 (both inclusive), return **false** else return **true**.
* If the month is not between 1 and 12 (both inclusive), return **false** else return **true**.
* If the day is not between 1 and 31 (both inclusive), return **false** else return **true**.

**Assumption:**

* All characters in the input string will be in UPPER CASE.

**class** **definitions**:

**class** **batch**

  ﻿      **method** **definitions**:

lengthCheck(String str):

**return** type: boolean

visibilty: **public**

        batchNumberCheck(String str):

**return** type: boolean

            visibilty: **public**

        yearCheck(String str)):

**return** type: boolean

            visibilty: **public**

        monthCheck(String str):

**return** type: boolean

            visibilty: **public**

        dayCheck(String str):

**return** type: boolean

            visibilty: **public**

        printBatchNumber(String str):

**return** type: String

            visibilty: **public**

        printDate(String str):

**return** type: String

            visibilty: **public**

**Methods**to be Implemented:

* **lengthCheck(String str):** Compute the length of string and return true if length is 12 else return false.
* **batchNumberCheck(String str):**Check if the batch number(First four digit/letters represents the batch number ) is correct according to the description provided, return true if correct else false.
* **yearCheck(String str)**: Check if the year( Between 2015 and 2020) is valid according to the description provided, return true if correct else false.
* **monthCheck(String str)**: Check if the month (Between 1 and 12) is valid according to the description provided, return true if correct else false.
* **dayCheck(String str)**: Check if the day( Between 1 and 31) is valid according to the description provided, return true if correct else false.
* **printBatchNumber(String str)**: Check if the batch number is valid according to the description provided, return batch number if valid else return **null.**
* **printDate(String str)**: Check if the date is valid according to the description provided, return date in string(Format: DD/MM/YYYY) if valid else return **null.**

**﻿**

**NOTE:**

* The argument **str** in the above methods is the**CODE (e.g.** BL7A20181201).
* The first 4-digit/letter in the batch number and last 8-digits represents the date in **YYYYMMDD**format.

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class Ingredient:

﻿data members:

String name

int qty

visibility : private

Ingredient(String name, int qty): constructor **with** **public** visibility

**Define** getters setters **with** **public** visibility

**class** Dish:

﻿**data** members:

ArrayList<Ingredient> ingredients

visibility : **public**

Dish(ArrayList<Ingredient> ingredients): **constructor** **with** **public** visibility

method definitions:

prepareDish(ArrayList<Ingredient> ingredientList):

**return** **type**: int

visibility: **public**

﻿

totalQty():

**return** **type**: int

visibility: **public**

**Task**

**Class Ingredient**

**-** define all the variables according to the above specifications.

- define a**constructor** with public visibility.

**Class Dish**

**-** define all the variables according to the above specifications.

- define a**constructor** with public visibility.

**Implement the below methods for this class:**

**-int** **prepareDish(ArrayList<Ingredient> ingredientList):**

* Write a code that validates and return int according to the below conditions -

1. If **all the ingredients in the ingredientList**are present in the list of **ingredients**with qty less,then return **1.**
2. **Else return -1.**

**-int** **totalQty():**

* Write a code that returns the **sum of the qty of all the ingredients** in the list **ingredients.**

**Sample Input**

ArrayList<Ingredient> ingredients = **new** ArrayList<>();

ingredients.**add**(**new** Ingredient("Tomato",100));

ingredients.**add**(**new** Ingredient("Potato",200));

Dish dish = **new** Dish(ingredients);

ArrayList<Ingredient> reqingredients = **new** ArrayList<>();

Ingredient ingredient11 = **new** Ingredient("Tomato",100);

reqingredients.**add**(ingredient11);

dish.prepareDish(reqingredients);

dish.totalQty();

**Sample Output**

1

300

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class Build:

data members:

int length

int width

visibility : public

Build(int length, int width): constructor **with** **public** visibility

﻿ method definition:

builder(int blength, int bwidth, int **count**) throws ShortageException, TendorException:

**return** : **String**

visibility : **public**

﻿

**class** ShortageException extends **Exception**:

method definitions:

ShortageException(**String** msg)

visibility: **public**

**class** TendorException extends **Exception**:

method definitions:

TendorException(**String** msg)

visibility: **public**

**Class Build**

**-** define all the variables according to the above specifications.

**-** define a **constructor** according to the above specifications.

**Implement the below methods for this class:**

**-String** **builder(int blength, int bwidth,int count) throws ShortageException, TendorException:**

* Write a code that accepts the length, width, and count of the bricks available and return the result according to the mentioned scenarios below -

1. If the**brick length(blength) evenly divides the parameter length, brick width(bwidth) evenly divides the parameter width, and the number of bricks required to cover the total area(length\*breadth) is less than equal to the parameter count** then return **"Builder!!".**
2. If the**brick length(blength) evenly divides the parameter length, brick width(bwidth) evenly divides the parameter width, and the number of bricks required to cover the total area(length\*breadth) is greater than the parameter count** then throw the **ShortageException**with the message **"Need more bricks".**
3. If the**brick length(blength) does not evenly divides the parameter length or brick width(bwidth) does not evenly divide the parameter width, and the number of bricks required to cover the total area(length\*breadth) is less than equal to the parameter count** then throw the **TendorException**with the message **"Building dimension mismatched".**
4. If the**brick length(blength) does not evenly divides the parameter length or brick width(bwidth) does not evenly divide the parameter width, and the number of bricks required to cover the total area(length\*breadth) is greater than the parameter count** then throw the **ShortageException**with the message **"Need more bricks with dimension mismatched".**

**Class ShortageException**

- define **ShortageException** class derived from the Exception class.

**Class** **TendorException**

- define **TendorException** class derived from the Exception class.

**Sample Input**

Build build = **new** Build(100, 100);

build.builder(10,10,100);

build.builder(10,10,50);

**Sample Output**

Builder!!

**ShortageException: Need more bricks﻿**

**Description**

Amit is a government employee. Indian government provides scholarships to college students on a performance basis. Amit has been given the responsibility to assign scholarships based on student percentage and find the total amount of scholarship.

Help Amit to complete the classesusing the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:

﻿class Student:

data members:

String name

String collegeName

float percentage

float scholarship

visibility: private

(name, collegeName, percentage): constructor **with** **public** visibility

**Define** getter setters **with** **public** visibility

**class** Portal:

**data** **member**:

ArrayList<Student> studentList

method definitions:

assignScholarship():

**return** **type**: void

﻿

totalScholarship():

﻿**return** **type**: float

﻿

totalMaxScholarshipOfCollege():

﻿**return** **type**: **String** ﻿

**Task**

Class **Student**

**-**define the **String** variable **name.**

**-**define the **String** variable **collegeName.**

**-**define the **float** variable **percentage.**

**-**define the **float** variable **scholarship.**

**-**define a **constructor** and **getter setters** according to the above specifications.

Class **Portal**

**-**define the **ArrayList<Student>** variable **studentList.**

**Implement the below methods for this class:**

**-void assignScholarship():**

* The scholarship is going to be assigned based on the percentage given below:
* If percentage >=91, scholarship = 10000.
* If percentage >= 81, scholarship = 5000.
* If percentage < 81, scholarship = 0.
* Set the scholarship according to the percentage in **studentList**.

**-float totalScholarship():**

* Write a code to find the total scholarship going to be paid by the government.
* Return the total scholarship.

Example:

**studentList** = [{"Steve", "IIT", 89, 5000}, {"Bob", "NIT", 94, 10000}, {"Alice", "Abcd", 59, 0}] , **totalScholarship** = 15000.

**-String totalMaxScholarshipOfCollege():**

* Write a code to find the total scholarship of all the colleges and return the college name with the maximum scholarship.

Example:

**studentList** = [{"Steve", "IIT", 89, 5000}, {"Bob", "NIT", 94, 10000}, {"Alice", "NIT", 85, 5000}] then the **collegeName** = "NIT".

**Sample Input**

Portal obj = **new** Portal();

obj.studentList.**add**(**new** Student("Steve", "IIT", 89));

obj.studentList.**add**(**new** Student("Bob", "NIT", 94));

obj.studentList.**add**(**new** Student("Alice", "Abcd", 59));

------------------------------------------------------------

obj.assignScholarship();

obj.totalScholarship();

obj.totalMaxScholarshipOfCollege();

**Sample Output**

15000.0

**NIT**

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class Stocks:

float stockRate

visibility : public

Stocks(float stockRate) : Constructor **with** **public** visibility

﻿method definition:

﻿checkRate(float Rate) throws **Exception**:

**return** **type**: **String**

visibility: **public**

﻿

**class** StockHighException extends **Exception**:

﻿method definition:

﻿ StockHighException(**String** msg)

visibility: **public**

﻿

**class** StockLowException extends **Exception**:

﻿method definition:

﻿ StockLowException(**String** msg)

visibility: **public** ﻿

**Task**

**Class** **Stocks**

-define all the data members as per the given specifications.

-define the constructor with public visibility.

**-Implement the below methods for this class:**

**-String** **checkRate(float rate) throws Exception:**

* Write a code that checks for the Profit and Loss in the stock rate**.**
* If the given parameter rate is 80% of the stockRate then return a string "Hope to raise".
* If the given parameter rate is less than 80% of the stockRate then throw StockLowException with a message "Under Loss".
* If the given parameter rate is greater than 80% of the stockRate then throw StockHighException with a message "Under Profit".

**Class** **StockHighException extends Exception:**

-Define**StockHighException** class derived from the Exception class.

**Class** **StockLowException extends Exception:**

-Define**StockLowException** class derived from the Exception class.

**Sample Input**

Stocks stock = **new** Stocks(10.0f);

stock.checkRate(5.0f);

**Sample Output**

**StockLowException** : Under Loss

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class WalkingBoy:

int ﻿stepSize

int blockSize

visibility : public

WalkingBoy(int stepSize, int blockSize) : Constructor **with** **public** visibility

﻿method definition:

﻿targetHit(**String** platform) throws **Exception**:

**return** **type**: **String**

visibility: **public**

﻿

**class** BombBlast extends **Exception**:

﻿method definition:

﻿ BombBlast(**String** msg)

visibility: **public** ﻿

**Task**

**Class** **WalkingBoy**

-define all the data members as per the given specifications.

-define the constructor with public visibility.

**-Implement the below methods for this class:**

**-String** **targetHit(String platform) throws Exception:**

* Write a code that checks whether a **boy hit the target/bomb or not.**
* **platform** is a string that contains alphanumeric values.**divide the platform into N block of length blockSize.** For every block**check if the first stepSize character contains x in it**then throw **BombBlast**Exception with a message **"You hit the target".**
* Else **return "Win".**

**Class** **BombBlast extends Exception:**

-Define**BombBlast** class derived from the Exception class

**Example for Reference**

*stepSize = 2, blockSize = 3, platform = "1212121x212"*

*After dividing the platform into N blocks of blockSize = 3 -> ["121", "212", "1x2", "12"]*

*After consedering only stepSize for every block -> ["12", "21", "1x", "12"]*

*3rd block contains x. therefore it throw****BombBlast****exception.*

**Sample Input**

WalkingBoy boy = **new** WalkingBoy(2,3);

boy.targetHit("1212121x212");

**Sample Output**

**BombBlast** : You hit the target

**Description**

**Case Study:**

Student Scholarship Scheme:

* By default, all students in a college will be assigned with a Scholarship scheme based on the score range of the student. Refer the below given table to find the eligible scholarship scheme specific to a student.

score | Scholarship Scheme

score > 95 | Scheme a

score >= 90 and score <= 95 | Scheme b

score < 90 | no scheme

**Task**

On the basis of above case study implement a class to accept multiple student details and store all student objects in a HashMap. The functionalities need to be implemented are as follows:

1. Add student details to HashMap.
2. Accept scholarship scheme and display student details based on scholarship scheme.
3. Delete student details from map.

**Note:** Refer the code stub for more clarity.

**Sample Input**

Student s= **new** Student("Alice", 10, 94);

70

11

**Sample Output**

Name: Alice Id: 10 Score: 94 ScholarshipScheme: scheme b

no scheme

false

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class Rating:

data members:

int imdbRating

int nominee

Rating(int imdbRating, int nominee): constructor **with** **public** visibility

**class** Validator:

method definitions:

canBeConsideredForTheAward(Rating rating) throws **Exception**:

**return** **type**: **String**

visibility: **public**

﻿

sendInvite(Rating rating) throws **Exception**:

**return** **type**: **String**

visibility: **public**

﻿

**class** MovieRatingException:

method definitions:

MovieRatingException(**String** msg)

visibility: **public** ﻿

**Task**

Class **Rating**

**-**define the **int**variable **imdbRating.**

**-**define the **int** variable **nominee**

**-**define a **constructor** according to the above specifications.

Class **Validator**

**Implement the below methods for this class:**

**-String** **canBeConsideredForTheAward(Rating rating) throws Exception:**

* Write a code to validate the criteria for getting the award.
* **throw a MovieRatingException**if **imdbRating** is less than **7**with the message "**Movie not eligible for Filmfare award**".
* **throw a MovieRatingException**if **nominee** is less than **4** with the message "**Minimum 4 nominee required**".
* If no above exception found then return a string message "**Considered for the award**".

**-String sendInvite(Rating rating):**

* Write a code to send an invite to the nominee.
* If **canBeConsideredForTheAward** method throws a **MovieRatingException** then return a message "**Not invited**".(Use try-catch block)
* If it throws any other exception then return a message "**other exception**".
* If no exception found then return a message "**Actors and Directors Invited**".

**Sample Input**

Rating rating = **new** Rating(9, 7);

Validator v = **new** Validator();

------------------------------------------------------

String s = v.canBeConsideredForTheAward(rating);

String t = v.sendInvite(rating);

s.toLowerCase();

t.toLowerCase();

**Sample Output**

considered **for** the award

actors **and** directors invited

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class Car:

﻿data members:

int carId

int speed

visibility : public

Car(int carId, int speed): constructor **with** **public** visibility

**class** CarRacing:

﻿**data** members:

ArrayList<Car> cars

int trackCount

visibility : **public**

﻿

CarRacing(int trackCount): **constructor** **with** **public** visibility

method definitions:

carEntry(Car car):

**return** **type**: **String**

visibility: **public**

﻿

getWinner(int trackLength):

**return** **type**: int

visibility: **public**

**Task**

**Class Car**

**-** define all the variables according to the above specifications.

- define a**constructor** with public visibility.

**Class CarRacing**

**-** define all the variables according to the above specifications.

**-** define

* participants as **ArrayList<Car> cars = new ArrayList<>();**

- define a**constructor** with public visibility.

**Implement the below methods for this class:**

**-String** **carEntry(Car car):**

* Write a code that adds the given **car**to the list of **cars**according to the given conditions.
* If the trackCount is equal to the length of the list **cars**then return **"No Space".**
* If the list **cars**have already an entry with the **same id** as the given parameter **car**then return **"Already exists".**
* If both the above condition does not satisfy then add the given parameter **car**into the list of **cars**and return **"Start practicing".**

**-int** **getWinner(int trackLength):**

* Write a code that returns the**carId of the winner.**
* If no entry exists then return **-1.**
* Else return the carId of the **car** with maximum **time.**
* The formula to calculate the **time = trackLength \* speed.**
* If two cars have the same time then return the carId which is added to the list of **cars**first.

**Sample Input**

Car car = **new** Car(1,100);

CarRacing carRacing = **new** CarRacing(2);

carRacing.carEntry(car);

**Sample Output**

**Start** practicing

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class User:

data members:

String userName

String password

visibility: private

Rating(String userName, String password): constructor **with** **public** visibility

**Define** getters **and** setters **with** **public** visibility

**class** Validator:

**data** members:

**User** **user** = null

Validator(**User** **user**): **constructor** **with** **public** visibility

method definitions:

checkCredentials(**String** userName,**String** **password**) throws **Exception**:

**return** **type**: **String**

visibility: **public**

﻿

logIn(**String** userName, **String** **password**) throws **Exception**:

**return** **type**: **String**

visibility: **public**

﻿

**class** UserNameException extends **Exception**:

method definition:

UserNameException(**String** msg)

visibility: **public**

**class** PasswordException extends **Exception**:

method definition:

PasswordException(**String** msg)

visibility: **public** ﻿

**Task**

Class **User**

**-**define the **String**variable **userName.**

**-**define the **String**variable **password.**

**-**define a **constructor** according to the above specifications.

Class **Validator**

-Initialize the user variable with null initially.

-Assign the value that is received from the constructor.

**Implement the below methods for this class:**

**-String** **checkCredentials(String userName, String password) throws Exception:**

* Write a code to validate the user credentials.
* Check if the userName passed is equal to the userName of the user object.
* If userName is mismatched then throw new **UserNameException** with the message "**Invalid user name".**
* If the password doesn't match with the password of the user object then throw a new PasswordException with the message **"Invalid password".**
* **If no exception is thrown then return "Good to go".**

**-String logIn((String userName, String password) throws Exception:**

* Write a code to help the user log in.
* If **checkCredentials()**method throws a **UserNameException** or **PasswordException**then returns a message "**Log in failed**".(Use try-catch block)
* If it throws any other exception then return a message "**Other exception**".
* If no exception is found then return a message "**Welcome**".

**Sample Input**

User user =**new** User("shoyab.idrsi","Password@123");

Validator v= **new** Validator(user);

String t = v.logIn("shoyab.idrsi","Password@123");

String s = v.logIn("shoyab.idrsi","Password@123");

t.toLowerCase();

s.toLowerCase();

**Sample Output**

**good** to go

welcome

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class Employee:

data members:

String name

String projectName

int workingHrs

int bonus

visibility : public

Employee(String name, String projectName, int workingHrs):constructor﻿

method definitions:

setBonus() :

return type : String

visibility : public﻿﻿

generateId() :

return type : String

visibility : public﻿﻿

Task

Class **Employee**

**-** define all the variables according to the above specifications.

**-** define a **constructor** according to the above specifications, **set bonus as 0** in the constructor.

**Implement the below methods for this class:**

**-String** **setBonus():**

* Write a code that validate if the project name is in **{"web","tech","hack","SD","PD"}** and workingHrs is greater than or equals to 30.
* If the above condition satisfies then initialize**bonus with workingHrs / 10 and**return a message**"Congrats".**
* If the condition is not satisfied then return **"Work hard"**

**-String** **generateId():**

* Write a code that generates the unique employee id according to the given description -

1. First of all, find the minimum length (X) among name and projectName.
2. Then generate a new string that has 2\*X element in the sequence name[0]projectname[0]name[1]......name[X]projectName[X].
3. Then append workingHrs at the end of the above-generated string.

**Sample Input**

Employee employee = **new** Employee("doselect", "hack", 30);

employee.generateId();

**Sample Output**

**dhoascek30**

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class Register:

﻿method definitions:

﻿checkCredentials(String email, String pass, String cpass) throws Exception:return type: String

visibility: public

﻿

class InvalidEmailException extends Exception:

﻿method definitions:

﻿InvalidEmailException(String msg)

visibility: public ﻿

class InvalidPasswordException extends Exception:

﻿method definitions:

﻿InvalidPasswordException(String msg)

visibility: public ﻿

class PasswordNotMatchException extends Exception:

﻿method definitions:

﻿ PasswordNotMatchException(String msg)

visibility: public ﻿

**Task**

class **Register**

**Implement the below methods for this class:**

**-String** **checkCredentials(String email, String pass, String cpass):**

* Write a code to validate credentials.
* Throw these exceptions **considering the conditions in the same sequence in which they are given** -

1. throw an **InvalidEmailException,** if the email does not contain **"@" and ".",**with the message **"Invalid Email".**
2. throw an **InvalidPasswordException**, if a **pass length is less than 6 characters,**with the message **"Invalid Password".**
3. throw a **PasswordNotMatchException**, if a**cpass is not equal to the valid pass,**with a message **"Password not match".**

* If the email contains **"@" and "."**in it and the pass is the same as cpass with a length greater than equal to 6 then return **"Registered"**.

*Refer the example for better understanding.*

*email1 = "myemail@email"*

*email2 = "myemail@email.com"*

*pass = "pass1234"*

*cpass = "pass123"*

*email1, pass and cpass gives****InvalidEmailException****as email validation fails and it has to be checked at first as per the given sequence.*

*email2, pass and cpass gives returns****Registered.***

class **InvalidEmailException**

-Define **InvalidEmailException** class derived from the Exception class

class **InvalidPasswordException**

-Define **InvalidPasswordException** class derived from the Exception class

class **PasswordNotMatchException**

-Define **PasswordNotMatchException** class derived from the Exception class

**Sample Input**

Register user = **new** Register();

user.checkCredentials("tushar@gmailcom","hiiiiii","hiiiiii");

**Sample Output**

**InvalidEmailException**: Invalid Email

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class Candidate:

data members:

String name

int totalRating

int totalContest

Candidate(String name, int totalRating, int totalContest): constructor **with** **public** visibility

**class** Validator:

method definitions:

eligible(Candidate details) throws **Exception**:

**return** **type**: **String**

visibility: **public**

﻿

sendInvite(Candidate details) throws **Exception**:

**return** **type**: **String**

visibility: **public**

﻿

**class** CriteriaMismatchException:

method definitions:

CriteriaMismatchException(**String** msg)

visibility: **public** ﻿

**Task**

Class **Candidate**

**-**define the **String** variable **name.**

**-**define the **int**variable **totalRating.**

**-**define the **int** variable **totalContest.**

**-**define a **constructor** according to the above specifications.

Class **Validator**

**Implement the below methods for this class:**

**-String** **eligible(Candidate details):**

* Write a code to validate the eligibility criteria.
* **throw a CriteriaMismatchException**if **totalRating** is less than 1000 with the message "**minimum 1000total rating is required**".
* **throw a CriteriaMismatchException**if **totalContest** is less than 10 with the message "**minimum 10** **contest participation is required**".
* If no above exception found then return a string message "**eligible candidate**".

**-String sendInvite(Candidate details):**

* Write a code to send an invite to the testlink.
* If an eligible method throws a **CriteriaMismatchException** then return a message "**candidate is not eligible**".(Use try-catch block)
* If it throws any other exception then return a message "**other exception**".
* If no exception found then return a message "**invitation send**".

**Sample Input**

Candidate data = new Candidate("Steve", 1020, 23);

Validator **check** = **new** Validator();

-----------------------------------------------------------

check.eligible(**data**);

check.sendInvite(**data**);

**Sample Output**

eligible candidate

invitation send

**Description**

Given a sentence **S,** composed of words separated by spaces where each word consists of alphabetical letters only.

Your task here is to convert the sentence to "**Buffalo*Latin****"* (a user-defined language similar to **Pig Latin**.)

Below are the ***rules*** of Buffalo Latin:

* If a word begins with a **vowel** (a, e, i, o, or u), append "**ma**" to the end of each word. For example, the word 'ate' becomes 'atema'.
* If a word begins with a **consonant**, remove the first letter and append it to the end, then add "**ma**". For example, the word 'sky' becomes 'kysmaa'.
* Add one letter '**a**' to the end of each word per its word index in the sentence, starting with **1**. For example, the first word gets "a" added to the end, the second word gets "aa" added to the end and so on.

Implement a **Java** code based on the following specifications. Note that your code should match the specifications in a precise manner. Consider default visibility of classes, data fields and methods unless mentioned otherwise.

**Specifications:**

class definitions:

 class Main:

  method definitons:

    toBuffaloLatin(String S, **Set**<Character> vowel):

**return** **type**: **String**

      visibilty: **public**

**union**(**Set**<**String**> a, **Set**<**String**> b):

**return** **type**: **Set**<**String**>

       visibilty: **public**

**Task:**

Create a**Main** class and implement below given methods:

* **toBuffaloLatin(String S, Set<Character> vowel):** Method to return the final sentence representing the conversion from **S**to ***Buffalo*** ***Latin***.
* **Set<String> union(Set<String> a, Set<String> b):** Method to perform **union** on Hashset's **a** and **b**.

**Sample Input:**

"Hi This Is Buffalo Language"

**Sample Output:**

**iHmaa** hisTmaaa Ismaaaa uffaloBmaaaaa anguageLmaaaaaa

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class Song:

data members:

String name

String runTime

String singer

Song(String nm,String rt, String sng): constructor with public visibility

class PlayList:

data members:

List<Song> songs = new ArrayList<>();

method definitions:

playSong(String currentTime,Song song):

return type : String

visibility : public

﻿﻿

addSong(Song song):

return type: String

visibility: public﻿﻿

Task

Class **Song**

**-** define all the below-mentioned variables with the specified specifications -

1. name- A string that defines the name of the song.
2. runTime- A string that stores the runtime in minutes and second separated by ':' (For example=> 10:30).
3. singer- A string that contains the name of the singer.

**-** define a **constructor** according to the above specifications.

Class **PlayList**

**-** define the class according to the above specifications and

**Implement the below methods for this class:**

**-String** **playSong(String currentTime, Song song):**

* Write a code that validates that the passed **currentTime is less than the runTime of the song.**
* **If the conditions are satisfied then return "Song is being continued".**
* And if the condition is not satisfied then return a message "**Song replayed**".

**-String** **addSong(Song song):**

* Write a code that checks if the current song is present or not.
* If the song is present return "**Cannot be added**" else add to the ArrayList and return "**Added**"**.**

**Sample Input**

PlayList lis=**new** PlayList();

Song s1=**new** Song("Venom", "4:58","Eminem");

lis.addSong(s1);

**Sample Output**

**Cannot** be added

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

**class** **definitions**:

﻿**class** **SpamDetector**:

﻿data members:

**int** primaryCount

**int** promotionCount

**int** updateCount

visibility : **private**

Define getter **and** setters **for** the data members.

SpamDetector(**int** primaryCount, **int** promotionCount, **int** promotionCount): constructor with **public** visibility

﻿ method definition:

addmessage(String email) throws SpamAttackException, SuspendException:

**return** : String

visibility : **public**

class SpamAttackException extends Exception:

method definition:

SpamAttackException(String message)

visibility: **public**

﻿

class SuspendException extends Exception:

method definition:

SuspendException(String message)

visibility: **public**

**Class SpamDetector**

**-** define all the variables and constructor according to the above specifications.

**Implement the below methods for this class:**

**-String** **addMessage(String email) throws SpamAttackException, SuspendException:**

* Write a code that update the email type count on the basis of following -

1. If the email contains any of the element in the list ["offer", "bonanza", "sale", "hurry"] than update promotionCount with 1
2. Else If the email contains any of the element in the list ["excel", "word", "ppt", "update"] than update updateCount with 1
3. Else update primaryCount by 1.

* After Updating Count Check below conditions -

1. If the promotionCount > updateCount > primaryCount then throw SuspendException with a message **("Account Suspended")**.
2. Else If the **promotionCount > ((primaryCount+updateCount)\*0.75)**then throw SpamAttackException with a message **("Account in Attack").**
3. Else return**"Safe".**

Class **SuspendException**

- define **SuspendException** class derived from the Exception class.

Class **SpamAttackException**

- define **SpamAttackException** class derived from the Exception class.

**Sample Input**

SpamDetector spamDetector = **new** SpamDetector(10,10,10);

spamDetector.addmessage("iphone 21")

**Sample Output**

**Safe**

**Description**

Complete the classes using the Specifications given below. Consider default visibility of classes, data fields, and methods unless mentioned otherwise.

**Specifications**

class definitions:﻿

class Email:

data members:

String receiver

String sender

String message

String subject

Email(String receiver, String sender, String message, String subject): constructor **with** **public** visibility

**class** ComposeEmail:

**data** members:

Email email

**String** sessionTime

ComposeEmail(Email email, **String** sessionTime) : **Constructor** **with** **public** visibility

method definitions:

clickCompose(**String** currentTime):

**return** **type**: **String**

visibility: **public**

validateEmailAddress(**String** address) :

**return** **type** : boolean

visibility : **public**﻿﻿

Task

Class **Email**

**-** define all the below-mentioned variables with the specified specifications -

1. receiver- A string of public visibility that has comma-separated email addresses.
2. sender - A string of public visibility that denotes the email address of the sender.
3. message - A string of public visibility that denotes the message send by the sender to the receiver(s).
4. subject - A string of public visibility that denotes the subject of the email.

**-** define a **constructor** according to the above specifications.

Class **ComposeEmail**

**-** define all the variables according to the above specifications.

**-** define a **constructor** according to the above specifications.

**Implement the below methods for this class:**

**-String** **clickCompose(String currentTime):**

* Write a code that validates the following conditions -

1. **currentTime is less than the sessionTime.**
2. **validateEmailAddress method returns true for receiver(s) email.**
3. **validateEmailAddress method returns true for sender email.**

* If all the conditions are satisfied then check for the following conditions -

1. if the**length of the message and the subject is greater than 0**then replace all the**group of whitespaces in the message with a single space and return "Mail send".**
2. If the **length of both(message and subject) equals 0** then returns a message **"Enter a valid subject/message".**

* And if one of the conditions is not satisfied then return a message " **Login Again**".

**-boolean** **validateEmailAddress(String address):**

* Write a code that **checks that all the emails (comma separated in the send parameter address) are valid.**
* A valid email is an email address that has **"@" and "."**both present in it.
* If the conditions satisfy all the email addresses then return **True** else return **False.**

**Sample Input**

Email email = new Email("e1@g.c,e2@gh.com","t@g.c", "hi/    myfan", "tyu");

ComposeEmail composeEmail = new ComposeEmail(email,"1234");

composeEmail.clickCompose("66");

**Sample Output**

**Mail** Send

**Description**

**Environment Specification & Instructions**

**Languages allowed**

* **Java7**
* **Java8**

**Build Expectation**

**Problem Statement**

Following is the structure you have to complete and implement. Remember, no need to write main function. To print anything call:

**write**.on()

I have implemented the ***class A*** for you by seeing the main function call and the output think and make the entries in the classes.

**class** **A**

{

**int** i=10;

**static**

{

write.on("I am static A");

}

{

write.on("I am {} A");

}

**public** **A**()

{

write.on("I am parameterised A");

}

**public** **A**(**int** i)

{

**this**.i=i;

write.on("I am parameterised constructor A");

}

**void** **function**()

{

write.on("I am function A");

}

}

**class** **B** **extends** **A**

{

}

**class** **C** **extends** **B**

{

**static** **class** **D**

{

}

**class** **E**

{

}

}

**class** **F** **extends** **C**

{

**class** **H** **extends** **D**

{

}

**class** **G** **extends** **E**

{

}

}

Following is the main function and the output for your help, and I have separated every line execution in the main with a “\_\_” that will help you to figure out the class parameters.

**public** **static** **void** **main**(String[] args)

{

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_1");

A a = **new** F();

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2");

a.function();

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_3");

write.**on**(a.i);

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_4");

a=**new** A();

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_5");

write.**on**(a.i);

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_6");

write.**on**(B.i);

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_7");

write.**on**(C.D.i);

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_8");

F f=**new** F();

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_9");

F.H b=f.new H();

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_10");

b.function();

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_11");

F.G c=f.new G();

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_12");

write.**on**(c.i);

System.**out**.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_13");

}

**Output**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*1

I am static A

I am static B

I am static C

I am static F

I am {} A

I am parameterised constructor A

I am {} B

I am constructor B

I am {} C

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*2

I am function A

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*3

30

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*4

I am {} A

I am parameterised A

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*5

10

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*6

20

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*7

50

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*8

I am {} A

I am parameterised constructor A

I am {} B

I am constructor B

I am {} C

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*9

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*10

I am function C.D

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*11

I am {} A

I am parameterised constructor A

I am {} B

I am constructor B

I am {} C

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*12

60

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***\_\_\_*13

------------------------------------------------------------------------------------------------------------------