1. **Set First and Last**

**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".

**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.

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

1. **Java :Portal for a Hotel**

**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

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

1. **Employee Service Implementation**

**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

**IMPORTANT:**

* If you want to test your program you can implement a **Main()** method given in the stub and you can use RUN CODE to test your **Main()**, provided you have made valid function calls with valid data required.

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

1. **Stone Processing**

**Description**

You are working on a project to process stones in a stone quarry using Java. Each stone has different characteristics, such as weight (in kilograms) and color. Your task is to write a Java program that uses the Stream API to perform various operations on a collection of stones. 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 Stone:

﻿data member:

double weight

String color

visibility: private

﻿

Stone(double weight, String color) : Define the constructor **with** **public** visibility

﻿ **Define** getters **for** **all** the **data** members **with** **public** visibility.

toString method has been defined **for** you **as** a part **of** the code stub.

**class** StoneProcessingApp:

﻿method definitions:

getTotalWeight(**List**<Stone> stones):

﻿ **return** **type**: **double**

visibility: **public**

getRedStone(**List**<Stone> stones):

﻿ **return** **type**: **List**<Stone>

visibility: **public**

﻿

getHeaviestStone(**List**<Stone> stones):

﻿ **return** **type**: Stone

visibility: **public**

**Task:**

**class Stone:**

**-**Define the class according to the above specifications

**class StoneProcessingApp:**

Implement the below method for this class using Stream API methods:

* **double getTotalWeight(List<Stone> stones):**Calculate and return the total weight of all stones.
* **List<Stone> getRedStone(List<Stone> stones):**Find and return all stones that are of a specific color ("red").
* **Stone getHeaviestStone(List<Stone> stones):**Find and return the heaviest stone in terms of weight.

**Sample Input**

List<Stone> stones = **new** ArrayList<>();

stones.**add**(**new** Stone(5.0, "red"));

stones.**add**(**new** Stone(8.0, "blue"));

stones.**add**(**new** Stone(6.5, "red"));

stones.**add**(**new** Stone(4.2, "green"));

stones.**add**(**new** Stone(7.8, "blue"));

StoneProcessingApp sta = **new** StoneProcessingApp();

sta.getTotalWeight(stones);

sta.getRedStone(stones);

sta.getHeaviestStone(stones);

**Sample Output**

31.5

[Stone{weight=5.0, color='red'}, Stone{weight=6.5, color='red'}]

Stone{weight=8.0, color='blue'}

**Note:**

You can make suitable function calls and use **the RUN CODE** button to check your **main()** method output.

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

1. **Parcel Management System**

**Description**

**Problem Statement**

You are tasked with implementing a simple Parcel Delivery System in Java. The system should be able to handle parcels, each identified by a unique parcel ID. Each parcel has sender and recipient names, an address, weight, and delivery status.

**Address Class:**

->Create a class named Address with attributes for street, city, state, and zip code, making them only accessible within the declared class.

->Initialize the data members for the object of the class.

->Override the toString method to return a formatted string representation of the address.(Follow the below format)

*street + ", " + city + ", " + state + ", " + zipCode*

**Parcel Class:**

->Create a class named Parcel with the following private attributes:

* parcelId (a unique identifier for each parcel, starting from 1 and incrementing with each new parcel)
* senderName
* recipientName
* address (an instance of the Address class)
* weight
* deliveryStatus

->Initialize the data members for the object of the class.

->Implement methods:

* **getParcelId()** to retrieve the parcel ID.
* **setParcelId(int parcelId)** to set a specific parcel ID.
* **updateStatus(String newStatus)** to update the delivery status of the parcel.
* Override the toString method to return a formatted string representation of the parcel, including all its details.(Follow the below format)

*"Parcel{" +*

*"parcelId=" + parcelId +*

*", senderName='" + senderName + '\'' +*

*", recipientName='" + recipientName + '\'' +*

*", address=" + address +*

*", weight=" + weight +*

*", deliveryStatus='" + deliveryStatus + '\'' +*

*'}'*

**ParcelDeliverySystem Class:**

->Create a class named ParcelDeliverySystem to manage a list of parcels.

->Initialize the data members for the object of the class.

->Implement methods:

* **addParcel(Parcel parcel)** to add a new parcel to the system.
* **updateParcelStatus(int parcelId, String newStatus)** to update the delivery status of a specific parcel.
* **viewParcelDetails(int parcelId)** to retrieve and display the details of a specific parcel.

**Implementation Requirements:**

* The system should ensure that each parcel has a unique identifier.
* The system should handle the addition of parcels, updating their delivery status, and viewing parcel details.
* The classes should have appropriate encapsulation and error handling (e.g., checking if a parcel with a given ID exists before updating its status).
* Demonstrate the functionality of the system by creating instances of parcels, adding them to the system, updating their status, and viewing their details.

**﻿Sample Input**

ParcelDeliverySystem deliverySystem = **new** ParcelDeliverySystem();

Address senderAddress = **new** Address("123 Main St", "Cityville", "State", "12345");

Address recipientAddress = **new** Address("456 Oak St", "Townsville", "State", "67890");

Parcel parcel1 = **new** Parcel("Sender1", "Recipient1", senderAddress, 2.5, "In Transit");

Parcel parcel2 = **new** Parcel("Sender2", "Recipient2", recipientAddress, 1.8, "Pending");

deliverySystem.addParcel(parcel1);

deliverySystem.viewParcelDetails(1);

deliverySystem.updateParcelStatus(2, "Delivered");

**Sample Output**

Parcel added successfully.

Parcel Details:Parcel{parcelId=1, senderName='Sender1', recipientName='Recipient1', address=123 Main St, Cityville, State, 12345, weight=2.5, deliveryStatus='In Transit'}

Parcel **not** found **with** ID 2

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

1. **Color Palette**

**Description**

**Problem Statement**

You are tasked with creating a custom palette generator in Java using EnumSet to manage colors in a color palette. The palette should be used to create various color schemes based on user preferences.

Here's what the problem entails:

**Color Enum:**

**->**Define an **enum** named **Colour** that includes a variety of colors. Each color should have a name and an associated **RGB** value.

**->**The enum defines various color **constants** along with their corresponding **java.awt.Color** values.

->The private final **Color color** field is set for each constant.

**->**The constructor initializes each **color** constant with its associated color.

**->**The**getColor()** method should retrieve the corresponding java.awt.Color object.

**ColorPalette Class:**

**->**Create a class named **ColorPalett**e that utilizes **EnumSet** to store a collection of colors.

**->**This class should include methods to:

->**void** **addColor(Colour color):**

* This method allows you to add a color of the Colour enum to the palette.
* It takes a Colour enum value as a parameter and adds it to the palette EnumSet.

**->EnumSet<Colour> generateTriadicScheme(Colour baseColor):**

* This method generates a triadic color scheme based on a given base color.
* Triadic color schemes are composed of three colors evenly spaced around the color wheel.
* It calculates the two triadic colors using the findTriadicColors private method.
* Returns an EnumSet containing the base color and its two triadic colors.

**->EnumSet<Colour>** **generateAnalogousScheme(Colour baseColor):**

* This method generates an analogous color scheme based on a given base color.
* Analogous color schemes are composed of colors that are adjacent on the color wheel.
* It calculates the two analogous colors using the findAnalogousColors private method.
* Returns an EnumSet containing the base color and its two analogous colors.

**->private Colour[] findTriadicColors(Colour baseColor):**

* This is a private helper method used to calculate the two triadic colors for a given base color.
* It takes the base color, calculates its index within the enum, and finds the two colors that are two and four positions away in the enum array.
* Returns an array of Colour containing the two triadic colors.

**->private Colour[] findAnalogousColors(Colour baseColor):**

* This is a private helper method used to calculate the two analogous colors for a given base color.
* Similar to the triadic case, it calculates the index of the base color and finds the colors that are one and five positions away in the enum array.
* Returns an array of Colour containing the two analogous colors.

*Remember that you can leverage the****java.awt.Color****class to represent colors using****RGB****values and perform color calculations.*

**Sample Input**

ColorPalette palette = new ColorPalette();

 palette.addColor(Colour.RED);

 palette.addColor(Colour.GREEN);

 palette.addColor(Colour.BLUE);

﻿

Colour baseColor = Colour.RED;

palette.generateTriadicScheme(baseColor);

palette.generateAnalogousScheme(baseColor);

**Sample Output**

[RED, BLUE, ORANGE]

[RED, GREEN, PURPLE]

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

1. **Seasonal Activities Organizer**

**Description**

**Problem Statement**

You are building an application that helps users organize their seasonal activities based on the weather conditions. You want to use **EnumSet** to efficiently manage the activities for each season.

**->**Create a **Season enum**that includes various types of seasons **SPRING**, **SUMMER**, **AUTUMN** (Fall), and **WINTER**.

**->**Create an **Activity** **enum** that includes various types of activities like **HIKING, SWIMMING, SKIING, PUMPKIN\_CARVING**

**class SeasonalActivityOrganizer**

**->**Your task is to implement the following:

**->**Create an **EnumSet** for each season to store the activities that are appropriate for that season.

**->getActivitiesForSeason(Season season):**

* This method takes a Season enum value as input and returns a set of Activity enum values that are suitable for the given season.

*The default case in the switch statement should include appropriate error handling for cases where an unknown or unsupported season value is provided.*

*("Unknown season: " + season)*

**->addActivityForSeason(Activity activity, Season season):**

* This method adds an Activity enum value to the set of activities appropriate for the specified season and then returns the updated set of activities for that season("Unknown season: " + season).

*The default case in the switch statement should include appropriate error handling for cases where an unknown or unsupported season value is provided("Unknown season: " + season).*

**->removeActivityFromAllSeasons(Activity activity):**

* This method removes an Activity enum value from the set of activities for all seasons if it exists and then returns the updated set of all activities.

**->getAllActivities():**

* This method returns a set containing all the Activity enum values from all seasons.

**﻿Sample Input**

SeasonalActivityOrganizer organizer = **new** SeasonalActivityOrganizer();

   organizer.addActivityForSeason(Activity.HIKING, Season.SPRING);

   organizer.addActivityForSeason(Activity.SWIMMING, Season.SUMMER);

   organizer.addActivityForSeason(Activity.SKIING, Season.WINTER);

organizer.getAllActivities()

organizer.getActivitiesForSeason(Season.SPRING)

organizer.getActivitiesForSeason(Season.SUMMER)

organizer.getActivitiesForSeason(Season.WINTER)

organizer.removeActivityFromAllSeasons(Activity.HIKING);

organizer.getActivitiesForSeason(Season.SPRING)

**Sample Output**

[HIKING, SWIMMING, SKIING]

[HIKING]

[SWIMMING]

[SKIING]

[]