Object Oriented Programming Innovative Assignment :

2CS502

Report on: Real Estate Management System

B. Tech. Semester III

Report Prepared By:

Param Desai

23BCE208

School of Engineering

Institute of Technology

Nirma University

Ahmedabad 382481

I. Introduction

The Real Estate Management System is a software project designed to simplify and facilitate property transactions for users interested in buying or renting properties. Built using Java, this system leverages Object-Oriented Programming (OOP) principles to model real-world interactions between properties and users (buyers and sellers). The system serves as an educational demonstration of OOP concepts like inheritance, polymorphism, encapsulation, and abstraction.

The main objectives of the project are:
- To provide a platform where users can view, buy, or rent properties in a straightforward manner.
- To illustrate the use of Java OOP concepts in creating a system that closely mimics real-world applications.

II. Design and Architecture

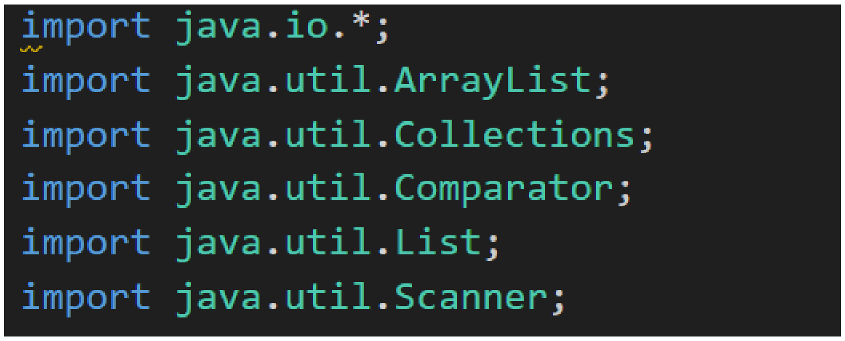
The implementation utilizes various Java topics effectively, showcasing the power of Object-Oriented Programming (OOP). Classes and objects define the `Property` and `Buyer` entities, encapsulating attributes through access modifiers to enhance data security. Inheritance is illustrated with `Apartment` and `Bungalow` extending the `Property` class, allowing shared functionality while maintaining unique characteristics. Constructors are employed to initialize the object state, while methods facilitate functionality, such as displaying property details. Polymorphism is demonstrated through the `displayPropertyDetails` method, overridden in subclasses to provide specific outputs. Collections, like `ArrayList`, manage multiple `Property` and `Buyer` instances, enabling dynamic storage and retrieval. Exception handling ensures robustness during property transactions, allowing graceful error management. Input/output operations read data from files for buyers and properties, ensuring that the application can function with external data sources. The use of the Stream API enhances filtering capabilities for property selection, making the buyer's experience more intuitive.

III. Concepts Used:

1. Object-Oriented Programming (OOP) concepts
2. Classes and Objects
3. Inheritance
4. Encapsulation
5. Abstraction
6. Constructors
7. Method Overloading
8. Method Overriding
9. Polymorphism
10. Abstract Classes
11. Exception Handling
12. Input/Output (I/O)
13. File Handling
14. Basic Data Types and Variables

IV. *Explanation of the code*

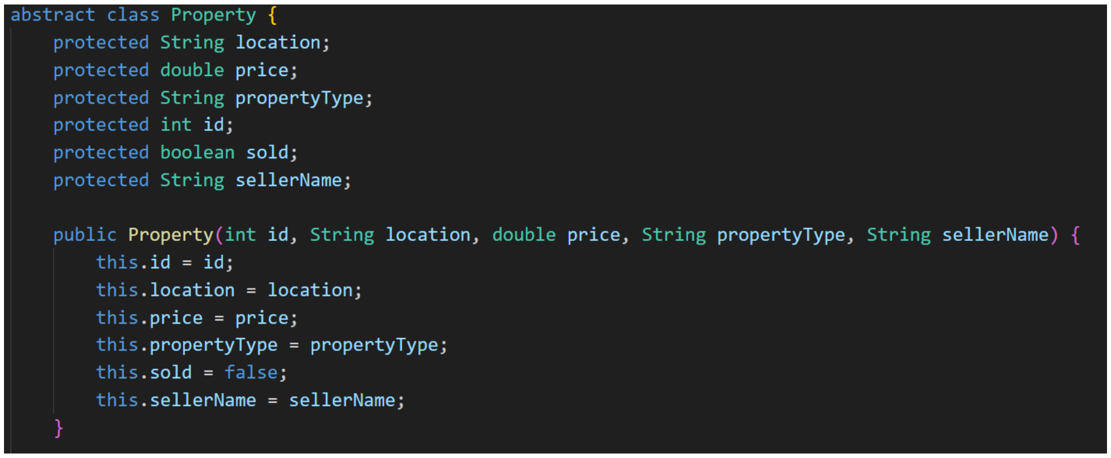
1. Package Imports:



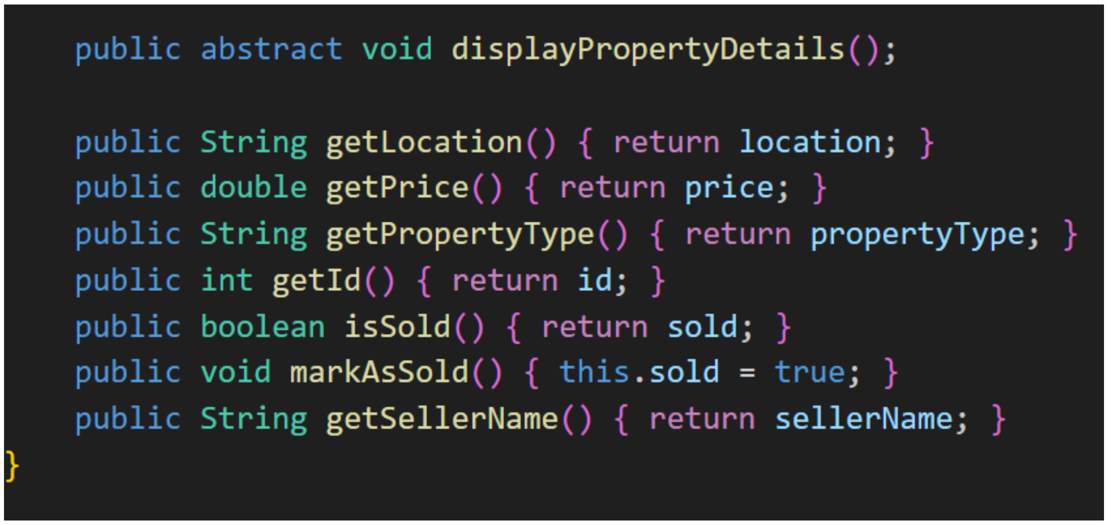
```java
import java.io.*;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.Scanner;
```

java.io: This package is used for input and output operations, allowing the program to read from files and handle exceptions related to I/O operations.
java.util: This package includes utility classes such as ArrayList, Collections, Comparator, List, and Scanner. These are essential for data structures, sorting, and taking user input.

2. Abstract Class: Property



```java
abstract class Property {
    protected String location;
    protected double price;
    protected String propertyType;
    protected int id;
    protected boolean sold;
    protected String sellerName;

    public Property(int id, String location, double price, String propertyType, String sellerName) {
        this.id = id;
        this.location = location;
        this.price = price;
        this.propertyType = propertyType;
        this.sold = false;
        this.sellerName = sellerName;
    }
```



```java
    public abstract void displayPropertyDetails();

    public String getLocation() { return location; }
    public double getPrice() { return price; }
    public String getPropertyType() { return propertyType; }
    public int getId() { return id; }
    public boolean isSold() { return sold; }
    public void markAsSold() { this.sold = true; }
    public String getSellerName() { return sellerName; }
}
```

Abstract Class: Property

The Property class is defined as an abstract class in the implementation, which serves as a foundational base for various types of property entities such as Apartment and Bungalow. An abstract class cannot be instantiated directly; instead, it provides a blueprint for its subclasses to follow, enforcing a common structure while allowing specific implementations.

Attributes:

- **location**: This attribute holds a string value representing the geographical location of the property. It is crucial for potential buyers, as it often influences the property's value and appeal.
- **price**: This double attribute specifies the cost of the property. The price is calculated based on various factors such as the property type and location, allowing for dynamic pricing models in the subclasses.
- **propertyType**: This string attribute indicates the category of the property, such as "Apartment" or "Bungalow". It

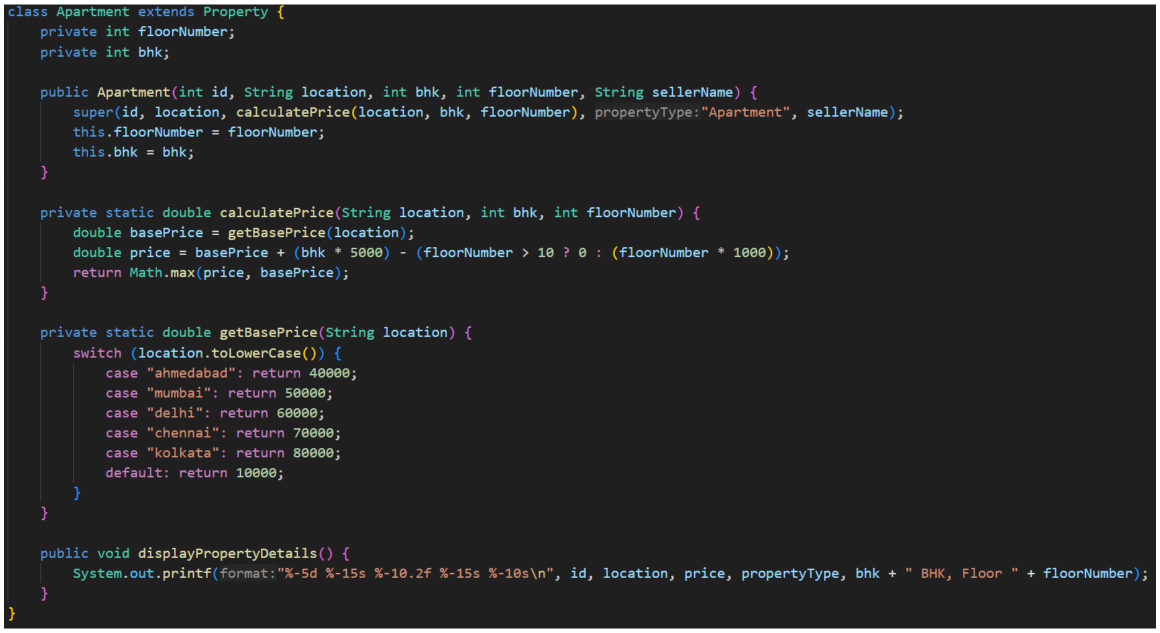
helps in classifying and filtering properties during operations like property search or suggestion for buyers.

- **id**: A unique integer identifier assigned to each property instance. This ID is essential for managing properties, allowing buyers and sellers to reference specific properties easily.
- **sold**: A boolean attribute that flags whether the property has been sold. This attribute is critical for tracking the availability of properties, ensuring that only unsold properties are presented to potential buyers.
- **sellerName**: This string attribute captures the name of the property's seller. It provides transparency in transactions and can be useful for buyers looking to contact sellers directly.

Constructor:
The constructor of the Property class initializes the property's attributes with the provided values. It sets the sold attribute to false, indicating that the property is available for sale upon instantiation. This constructor ensures that every property object is properly initialized with all necessary information right from the start.

Abstract Method:
The Property class defines an abstract method named displayPropertyDetails(). This method is intended to be implemented by each subclass to provide specific details about the property type it represents. By enforcing the implementation of this method in the subclasses, the Property class ensures that all derived property types have a consistent way of displaying their details.

3. Class: Apartment



```java
class Apartment extends Property {
    private int floorNumber;
    private int bhk;

    public Apartment(int id, String location, int bhk, int floorNumber, String sellerName) {
        super(id, location, calculatePrice(location, bhk, floorNumber), propertyType:"Apartment", sellerName);
        this.floorNumber = floorNumber;
        this.bhk = bhk;
    }

    private static double calculatePrice(String location, int bhk, int floorNumber) {
        double basePrice = getBasePrice(location);
        double price = basePrice + (bhk * 5000) - (floorNumber > 10 ? 0 : (floorNumber * 1000));
        return Math.max(price, basePrice);
    }

    private static double getBasePrice(String location) {
        switch (location.toLowerCase()) {
            case "ahmedabad": return 40000;
            case "mumbai": return 50000;
            case "delhi": return 60000;
            case "chennai": return 70000;
            case "kolkata": return 80000;
            default: return 10000;
        }
    }

    public void displayPropertyDetails() {
        System.out.printf(format:"%-5d %-15s %-10.2f %-15s %-10s\n", id, location, price, propertyType, bhk + " BHK, Floor " + floorNumber);
    }
}
```

The Apartment class is a subclass of the Property class, which means it inherits all the properties and methods defined in the Property class. This inheritance allows the Apartment class to reuse code and adhere to the established structure for properties, while also enabling it to introduce its specific attributes and behaviors.

Attributes:

- **floorNumber**: This integer attribute represents the specific floor on which the apartment is located. The floor number is significant in determining the desirability and market value of the apartment, as higher floors are often preferred for their views and reduced noise from street level.
- **bhk**: This integer attribute indicates the number of bedrooms, hall, and kitchen (BHK) present in the apartment. The BHK count is a critical factor in apartment listings, as it directly influences potential buyers' interest based on their needs for space and comfort.

Constructor:
The constructor of the Apartment class is responsible for initializing the apartment object with its unique attributes. It also calls a static method to calculate the price of the apartment based on its specific characteristics. This constructor ensures that each Apartment object is fully configured with its floor number and BHK count, along with a price that reflects its value according to the criteria set in the pricing methods.

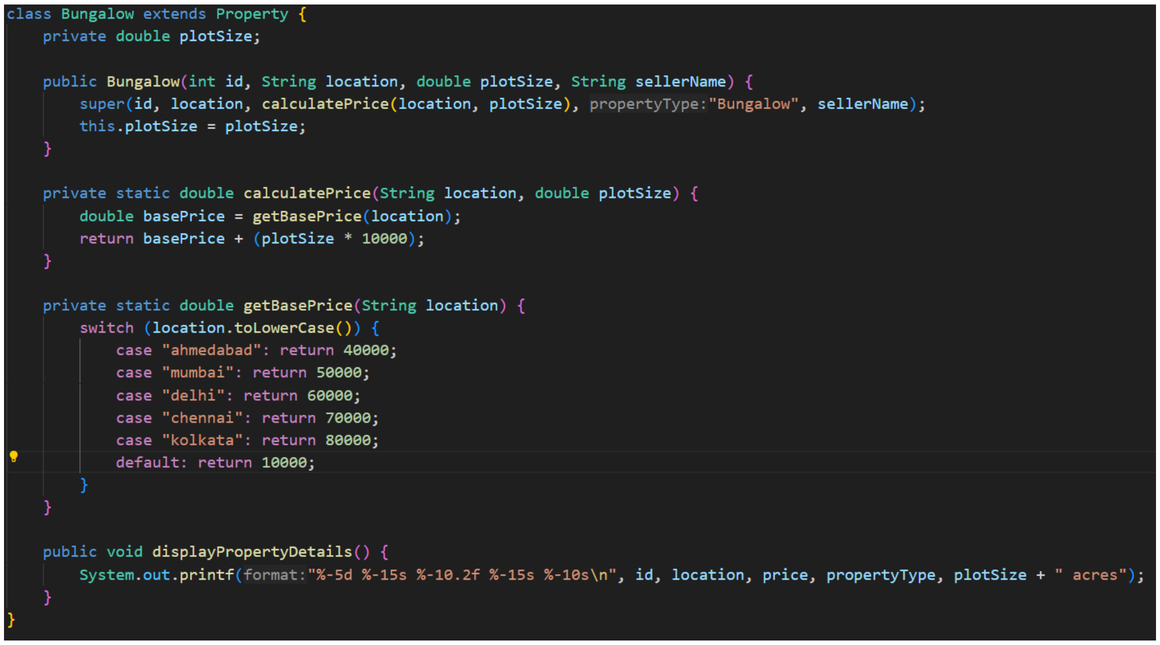
Price Calculation:

- **calculatePrice()**: This static method is key to determining the price of the apartment. It factors in the apartment's location, the number of BHK, and the floor number. Interestingly, higher floor numbers tend to decrease the price unless the apartment is located above the 10th floor, where the value may stabilize or even increase due to better views and amenities. This method encapsulates the logic for price determination, ensuring that pricing is consistent and can be easily adjusted if business rules change.
- **getBasePrice()**: This static method provides a base price for apartments based on the location. It returns different base values for various cities, reflecting the real estate market dynamics. By separating this logic into a dedicated method, the code promotes maintainability and clarity, allowing changes to base pricing strategies without affecting other parts of the class.

Method:

- **displayPropertyDetails()**: The displayPropertyDetails() method overrides the abstract method from the Property class. This implementation is tailored to output the specific details of the apartment in a formatted manner. It presents the apartment’s ID, location, calculated price, type, and unique details such as the BHK count and floor number. This method enhances the user experience by providing clear and accessible information about the apartment, aiding potential buyers in their decision-making process.

4. Class: Bungalow



```java
class Bungalow extends Property {
    private double plotSize;

    public Bungalow(int id, String location, double plotSize, String sellerName) {
        super(id, location, calculatePrice(location, plotSize), propertyType:"Bungalow", sellerName);
        this.plotSize = plotSize;
    }

    private static double calculatePrice(String location, double plotSize) {
        double basePrice = getBasePrice(location);
        return basePrice + (plotSize * 10000);
    }

    private static double getBasePrice(String location) {
        switch (location.toLowerCase()) {
            case "ahmedabad": return 40000;
            case "mumbai": return 50000;
            case "delhi": return 60000;
            case "chennai": return 70000;
            case "kolkata": return 80000;
            default: return 10000;
        }
    }

    public void displayPropertyDetails() {
        System.out.printf(format:"%-5d %-15s %-10.2f %-15s %-10s\n", id, location, price, propertyType, plotSize + " acres");
    }
}
```

The Bungalow class is a subclass of the Property class, which means it inherits the attributes and methods defined in the Property class. This design pattern allows the Bungalow class to leverage the existing structure and functionality associated with properties while introducing its own specific attributes and behaviors relevant to bungalow-type properties.

Attributes:

- **plotSize**: This double attribute represents the size of the land on which the bungalow is built. The plot size is an essential factor in real estate as it significantly impacts the property’s value. Bungalows typically occupy larger plots than apartments, and this attribute allows for the assessment of the property's potential for expansion, outdoor space, and overall marketability.

Constructor:

The constructor of the Bungalow class is designed to initialize the bungalow object with its unique attributes, including the plot size. Additionally, it calls a static method to calculate the price of the bungalow based on its specific characteristics. This constructor ensures that each Bungalow object is completely initialized with a plot size and a calculated price that reflects its value in the context of its location and size.

Price Calculation:

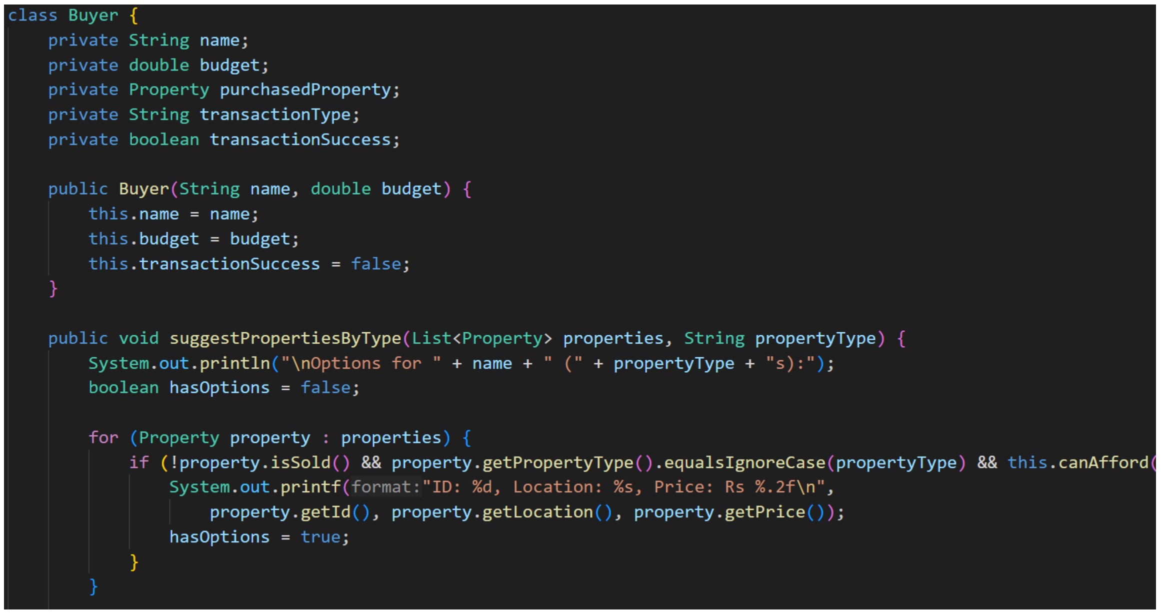
- **calculatePrice()**: This static method is crucial for determining the price of the bungalow. Unlike the Apartment class, which calculates price based on various factors including BHK and floor number, the calculatePrice() method for the Bungalow focuses primarily on the plot size. It adds a variable amount to a base price depending on the size of the land. This method encapsulates the pricing logic specific to bungalows, ensuring that the pricing model is clear.
- **getBasePrice()**: Similar to the Apartment class, the Bungalow class includes a getBasePrice() method that returns a base price determined by the location. This method is essential in setting a starting point for price

calculations based on the market dynamics of different cities. By isolating the base price logic in a dedicated method, the code enhances clarity and allows for straightforward adjustments to pricing strategies.

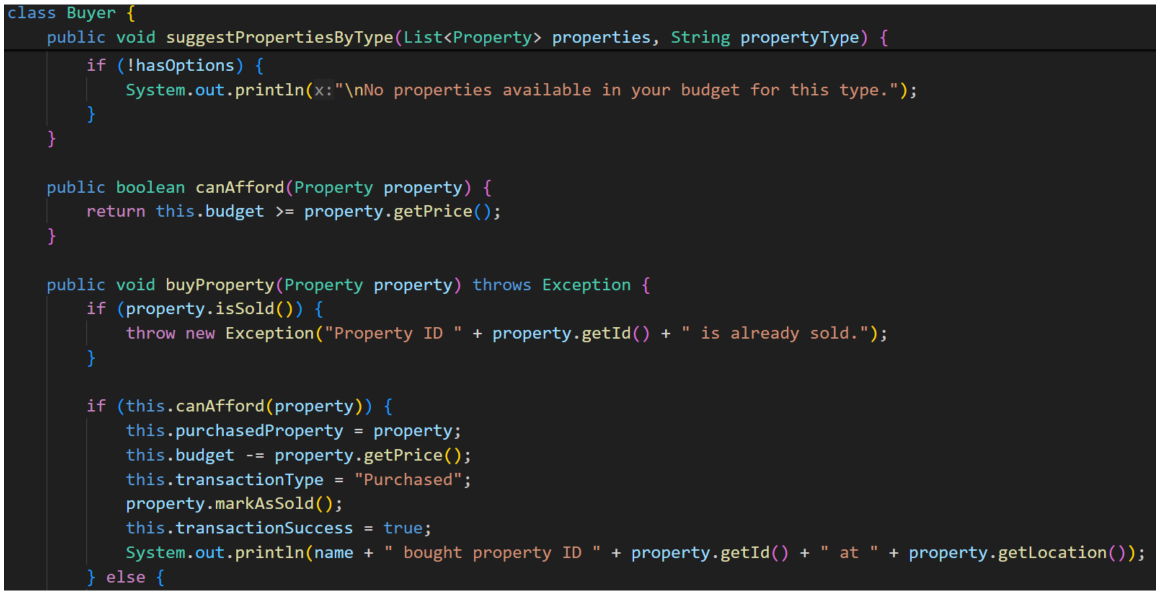
Method:

- **displayPropertyDetails()**: The displayPropertyDetails() method in the Bungalow class overrides the abstract method from the Property class, specifically tailored to output the details of the bungalow in a clear and formatted manner. This method presents essential information including the bungalow’s ID, location, calculated price, type, and its unique detail of plot size (expressed in acres). By providing a specific implementation for this method, the class enhances the user experience, making it easier for potential buyers to understand the unique aspects of the bungalow.

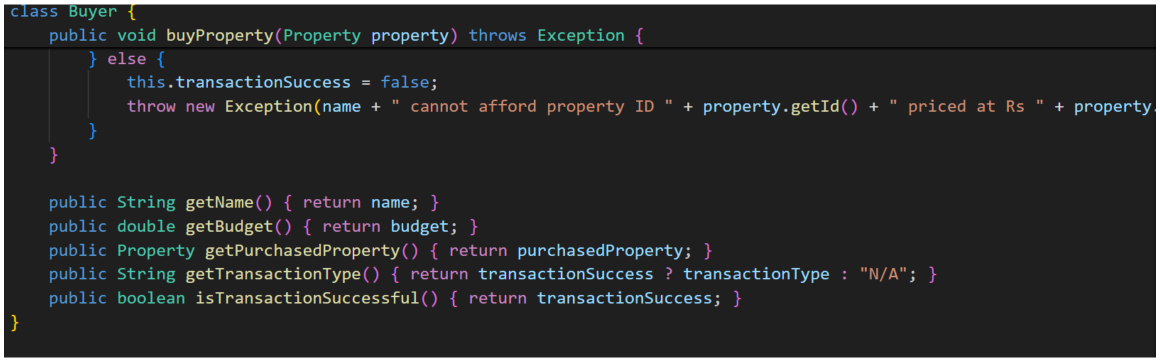
5. Class: Buyer



```java
class Buyer {
    private String name;
    private double budget;
    private Property purchasedProperty;
    private String transactionType;
    private boolean transactionSuccess;

    public Buyer(String name, double budget) {
        this.name = name;
        this.budget = budget;
        this.transactionSuccess = false;
    }

    public void suggestPropertiesByType(List<Property> properties, String propertyType) {
        System.out.println("\nOptions for " + name + " (" + propertyType + "s):");
        boolean hasOptions = false;

        for (Property property : properties) {
            if (!property.isSold() && property.getPropertyType().equalsIgnoreCase(propertyType) && this.canAfford(
                System.out.printf(format:"ID: %d, Location: %s, Price: Rs %.2f\n",
                    property.getId(), property.getLocation(), property.getPrice());
                hasOptions = true;
            }
        }
```



```
class Buyer {
    public void suggestPropertiesByType(List<Property> properties, String propertyType) {
        if (!hasOptions) {
            System.out.println(x:"\nNo properties available in your budget for this type.");
        }
    }

    public boolean canAfford(Property property) {
        return this.budget >= property.getPrice();
    }

    public void buyProperty(Property property) throws Exception {
        if (property.isSold()) {
            throw new Exception("Property ID " + property.getId() + " is already sold.");
        }

        if (this.canAfford(property)) {
            this.purchasedProperty = property;
            this.budget -= property.getPrice();
            this.transactionType = "Purchased";
            property.markAsSold();
            this.transactionSuccess = true;
            System.out.println(name + " bought property ID " + property.getId() + " at " + property.getLocation());
        } else {
```



```
class Buyer {
    public void buyProperty(Property property) throws Exception {
        } else {
            this.transactionSuccess = false;
            throw new Exception(name + " cannot afford property ID " + property.getId() + " priced at Rs " + property.
        }
    }

    public String getName() { return name; }
    public double getBudget() { return budget; }
    public Property getPurchasedProperty() { return purchasedProperty; }
    public String getTransactionType() { return transactionSuccess ? transactionType : "N/A"; }
    public boolean isTransactionSuccessful() { return transactionSuccess; }
}
```

The Buyer class encapsulates the characteristics and behaviors of a buyer in the property market. This class is pivotal in managing buyer-specific operations, allowing buyers to search for properties that fit their needs, manage their budgets, and conduct transactions. By structuring this class effectively, the implementation facilitates the interaction between buyers and properties, promoting a user-friendly experience in the property management system.

Attributes:

- **name**: This attribute represents the buyer's name as a string. It is essential for personal identification and interaction within the system. The name attribute helps to create a more personalized experience, allowing the system to address buyers directly when displaying property options and transaction details.

- **budget**: The budget attribute, represented as a double, signifies the maximum amount of money that the buyer is willing and able to spend on purchasing a property. This attribute is critical for ensuring that the buyer does not consider properties that exceed their financial means, thereby streamlining the selection process and enhancing user satisfaction.
- **purchasedProperty**: This attribute holds a reference to the specific property that the buyer has purchased. By maintaining this link, the class can track what the buyer has acquired and facilitate further actions such as displaying transaction details or managing post-purchase inquiries.
- **transactionType**: The transactionType attribute is a string that indicates the nature of the transaction (e.g., purchased). This attribute helps in categorizing the types of interactions the buyer has with the property management system, providing insight into the buyer's activity and preferences.
- **transactionSuccess**: This boolean flag indicates whether the last transaction made by the buyer was successful. It is essential for handling transaction logic within the system, allowing for appropriate user feedback and transaction history tracking.

Constructor:

The constructor of the Buyer class initializes the buyer's name and budget. It sets the initial state of the buyer object when it is created, ensuring that all necessary attributes are populated. This constructor enhances the clarity of the code by explicitly defining how a buyer is instantiated, while also ensuring that the budget and name are validated and correctly formatted before being assigned.

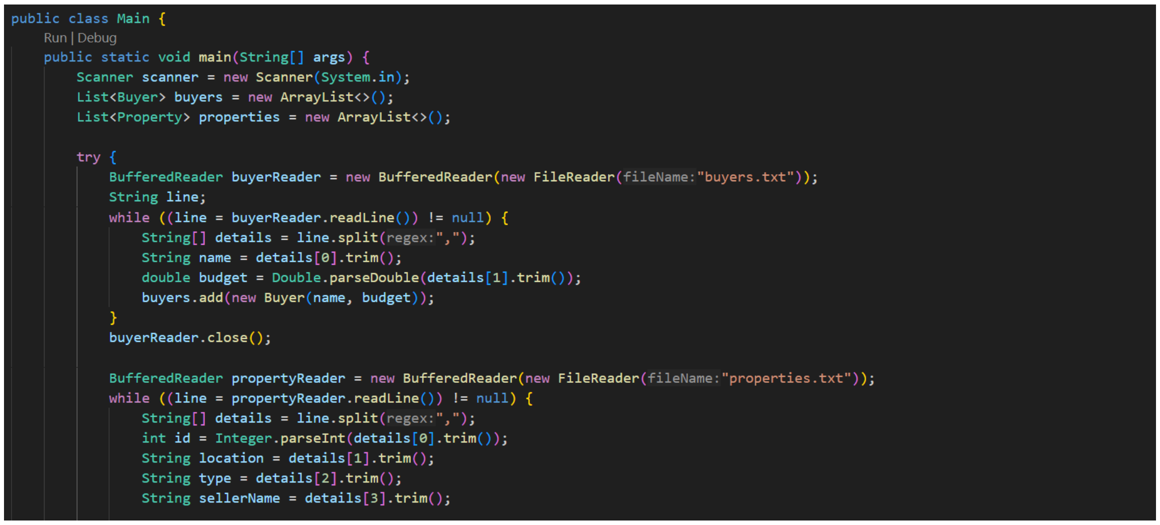
Methods:

- **suggestPropertiesByType(List<Property> properties, String propertyType)**: This method takes a list of properties and a specified property type as arguments. It filters through the properties to find those that are unsold and fit within the buyer's budget. By using this method, the buyer can receive tailored property suggestions based on their preferences and financial constraints. The method also handles the case where no suitable properties are available, providing appropriate feedback to the user. This functionality enhances the buyer's experience by simplifying the property search process.
- **canAfford(Property property)**: The canAfford() method checks if the buyer's current budget is sufficient to purchase a given property. By returning a boolean value, this method allows other components of the system to make informed decisions regarding the buyer’s ability to proceed with a transaction. This check prevents errors and enhances the robustness of the system by ensuring that transactions are only processed when financially viable.
- **buyProperty(Property property)**: This method handles the entire purchasing process for a property. It performs several critical operations:
    - **Check if the property is sold**: The method first verifies whether the selected property is already sold. If it is, an exception is thrown, providing immediate feedback to the buyer.

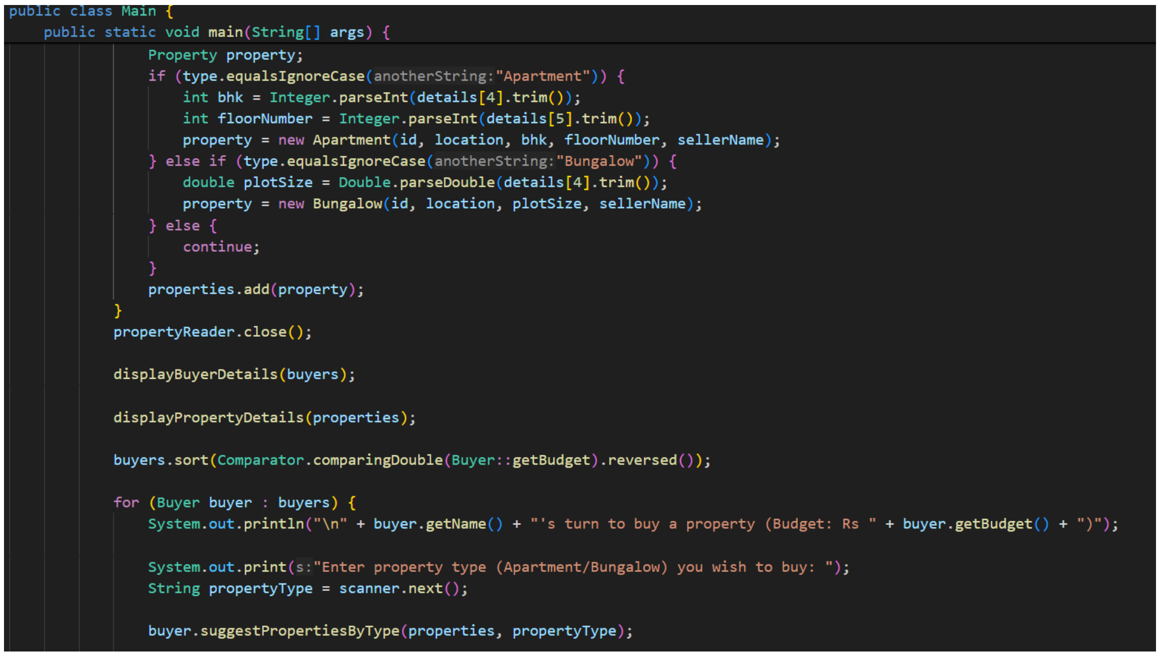
- **Affordability check**: It then checks if the buyer can afford the property using the canAfford() method. If the buyer's budget is insufficient, an exception is raised, indicating the specific reason for the failure to purchase.
- **Transaction execution**: If both checks pass, the method proceeds to update the buyer’s budget by deducting the property price, marking the property as sold, and setting the purchasedProperty attribute to the acquired property. It also updates the transactionType and transactionSuccess attributes accordingly.

This comprehensive handling of the purchasing logic not only ensures a smooth transaction process but also maintains data integrity throughout the system by effectively managing state changes and error handling.

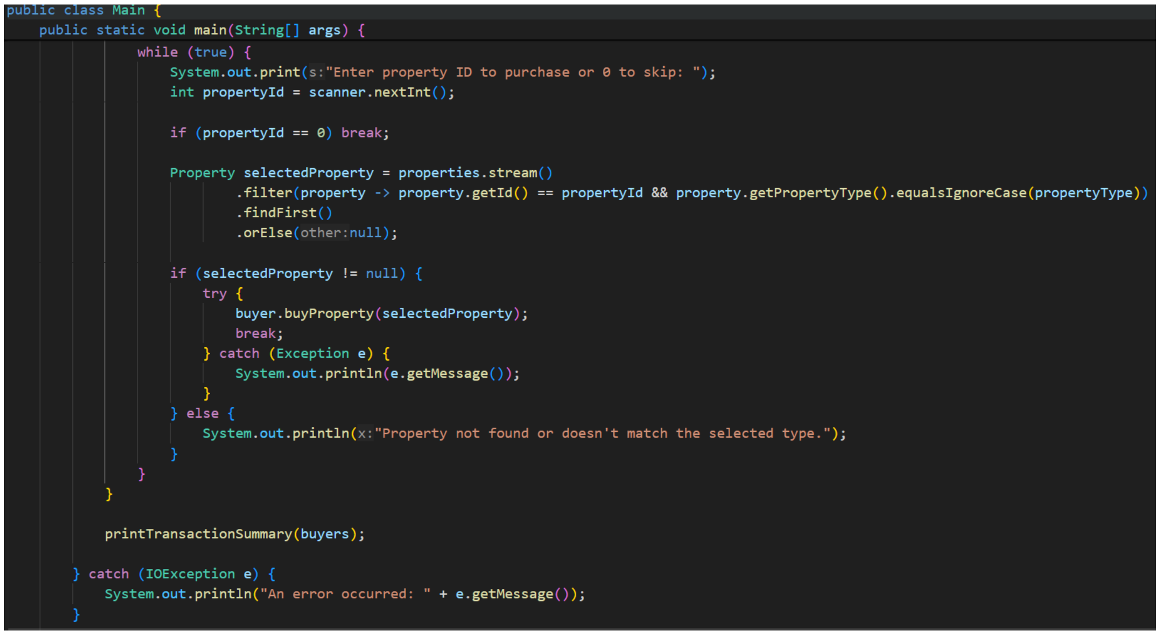
6. Main Class



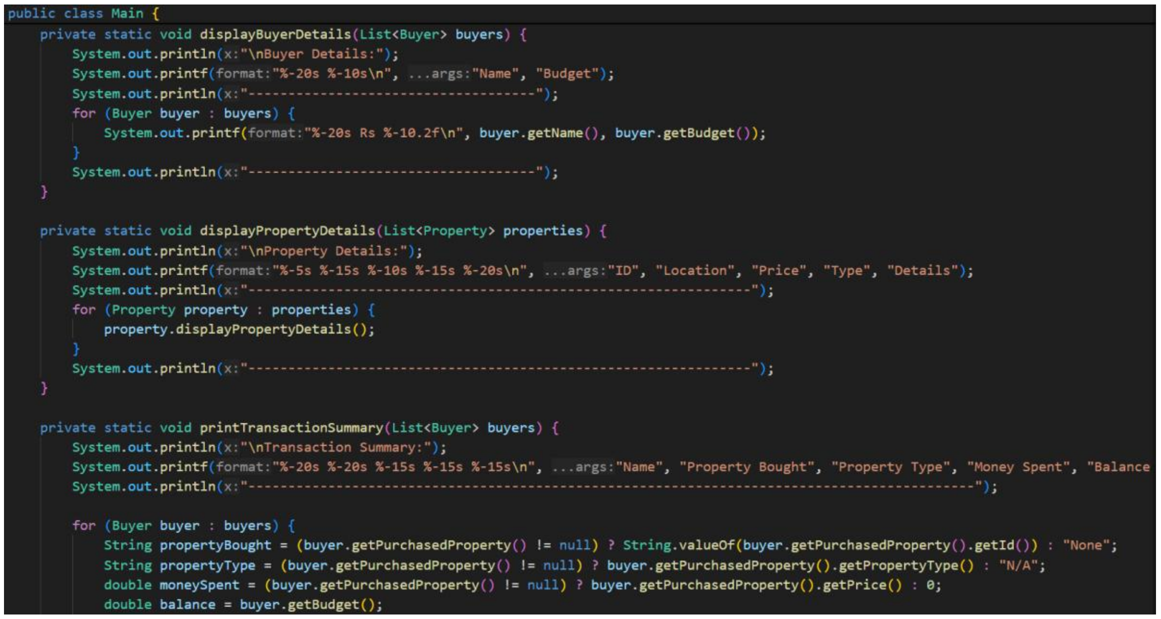
```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        List<Buyer> buyers = new ArrayList<>();
        List<Property> properties = new ArrayList<>();

        try {
            BufferedReader buyerReader = new BufferedReader(new FileReader(fileName:"buyers.txt"));
            String line;
            while ((line = buyerReader.readLine()) != null) {
                String[] details = line.split(regex:",");
                String name = details[0].trim();
                double budget = Double.parseDouble(details[1].trim());
                buyers.add(new Buyer(name, budget));
            }
            buyerReader.close();

            BufferedReader propertyReader = new BufferedReader(new FileReader(fileName:"properties.txt"));
            while ((line = propertyReader.readLine()) != null) {
                String[] details = line.split(regex:",");
                int id = Integer.parseInt(details[0].trim());
                String location = details[1].trim();
                String type = details[2].trim();
                String sellerName = details[3].trim();
```



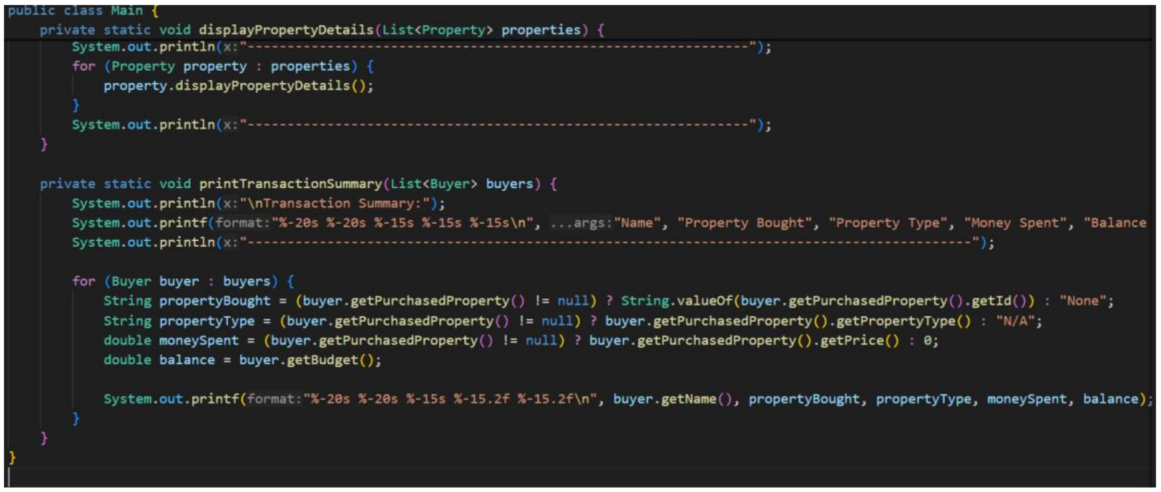
```java
public class Main {
    public static void main(String[] args) {
                Property property;
                if (type.equalsIgnoreCase(anotherString:"Apartment")) {
                    int bhk = Integer.parseInt(details[4].trim());
                    int floorNumber = Integer.parseInt(details[5].trim());
                    property = new Apartment(id, location, bhk, floorNumber, sellerName);
                } else if (type.equalsIgnoreCase(anotherString:"Bungalow")) {
                    double plotSize = Double.parseDouble(details[4].trim());
                    property = new Bungalow(id, location, plotSize, sellerName);
                } else {
                    continue;
                }
                properties.add(property);
            }
            propertyReader.close();

            displayBuyerDetails(buyers);

            displayPropertyDetails(properties);

            buyers.sort(Comparator.comparingDouble(Buyer::getBudget).reversed());

            for (Buyer buyer : buyers) {
                System.out.println("\n" + buyer.getName() + "'s turn to buy a property (Budget: Rs " + buyer.getBudget() + ")");

                System.out.print(s:"Enter property type (Apartment/Bungalow) you wish to buy: ");
                String propertyType = scanner.next();

                buyer.suggestPropertiesByType(properties, propertyType);
```



```java
public class Main {
    public static void main(String[] args) {
                while (true) {
                    System.out.print(s:"Enter property ID to purchase or 0 to skip: ");
                    int propertyId = scanner.nextInt();

                    if (propertyId == 0) break;

                    Property selectedProperty = properties.stream()
                            .filter(property -> property.getId() == propertyId && property.getPropertyType().equalsIgnoreCase(propertyType))
                            .findFirst()
                            .orElse(other:null);

                    if (selectedProperty != null) {
                        try {
                            buyer.buyProperty(selectedProperty);
                            break;
                        } catch (Exception e) {
                            System.out.println(e.getMessage());
                        }
                    } else {
                        System.out.println(x:"Property not found or doesn't match the selected type.");
                    }
                }
            }

            printTransactionSummary(buyers);

        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
```



```java
public class Main {
    private static void displayBuyerDetails(List<Buyer> buyers) {
        System.out.println(x:"\nBuyer Details:");
        System.out.printf(format:"%-20s %-10s\n", ...args:"Name", "Budget");
        System.out.println(x:"-----------------------------------");
        for (Buyer buyer : buyers) {
            System.out.printf(format:"%-20s Rs %-10.2f\n", buyer.getName(), buyer.getBudget());
        }
        System.out.println(x:"-----------------------------------");
    }

    private static void displayPropertyDetails(List<Property> properties) {
        System.out.println(x:"\nProperty Details:");
        System.out.printf(format:"%-5s %-15s %-10s %-15s %-20s\n", ...args:"ID", "Location", "Price", "Type", "Details");
        System.out.println(x:"---------------------------------------------------------------");
        for (Property property : properties) {
            property.displayPropertyDetails();
        }
        System.out.println(x:"---------------------------------------------------------------");
    }

    private static void printTransactionSummary(List<Buyer> buyers) {
        System.out.println(x:"\nTransaction Summary:");
        System.out.printf(format:"%-20s %-20s %-15s %-15s %-15s\n", ...args:"Name", "Property Bought", "Property Type", "Money Spent", "Balance
        System.out.println(x:"---------------------------------------------------------------------------------------------");

        for (Buyer buyer : buyers) {
            String propertyBought = (buyer.getPurchasedProperty() != null) ? String.valueOf(buyer.getPurchasedProperty().getId()) : "None";
            String propertyType = (buyer.getPurchasedProperty() != null) ? buyer.getPurchasedProperty().getPropertyType() : "N/A";
            double moneySpent = (buyer.getPurchasedProperty() != null) ? buyer.getPurchasedProperty().getPrice() : 0;
            double balance = buyer.getBudget();
```



```java
public class Main {
    private static void displayPropertyDetails(List<Property> properties) {
        System.out.println(x:"---------------------------------------------------------------");
        for (Property property : properties) {
            property.displayPropertyDetails();
        }
        System.out.println(x:"---------------------------------------------------------------");
    }

    private static void printTransactionSummary(List<Buyer> buyers) {
        System.out.println(x:"\nTransaction Summary:");
        System.out.printf(format:"%-20s %-20s %-15s %-15s %-15s\n", ...args:"Name", "Property Bought", "Property Type", "Money Spent", "Balance
        System.out.println(x:"---------------------------------------------------------------------------------------------");

        for (Buyer buyer : buyers) {
            String propertyBought = (buyer.getPurchasedProperty() != null) ? String.valueOf(buyer.getPurchasedProperty().getId()) : "None";
            String propertyType = (buyer.getPurchasedProperty() != null) ? buyer.getPurchasedProperty().getPropertyType() : "N/A";
            double moneySpent = (buyer.getPurchasedProperty() != null) ? buyer.getPurchasedProperty().getPrice() : 0;
            double balance = buyer.getBudget();

            System.out.printf(format:"%-20s %-20s %-15s %-15.2f %-15.2f\n", buyer.getName(), propertyBought, propertyType, moneySpent, balance);
        }
    }
}
```

The Main class serves as the primary entry point for the property management system, orchestrating the overall functionality of the application. It brings together the various components, facilitating the interaction between buyers and properties and enabling smooth transactions within the real estate marketplace.

Variables:

- **scanner**: This variable is an instance of the Scanner class, utilized for reading user input from the console. The Scanner is essential for creating an interactive user experience, allowing buyers to input their preferences and select properties. It streamlines the process of gathering user input and enables dynamic interactions during the execution of the program.
- **buyers**: The buyers variable is a list that stores instances of the Buyer class. This collection enables the program to manage multiple buyers, keeping track of their individual details, such as names and budgets. By using a list, the application can easily add, remove, or manipulate buyer instances as needed, allowing for flexible management of buyers throughout the program's lifecycle.
- **properties**: Similarly, the properties variable is a list that holds instances of the Property class and its subclasses (Apartment and Bungalow). This collection allows the program to maintain a comprehensive inventory of available properties, facilitating the buying process for users. The ability to store multiple property instances enables the application to handle various types of properties efficiently and to provide tailored recommendations to buyers based on their preferences.

File Reading:

The Main class utilizes BufferedReader to read data from external text files (buyers.txt and properties.txt). This approach enables the application to load predefined buyer and property information, enhancing usability and efficiency. Each line of the file is parsed to extract relevant details, such as buyer names, budgets, property IDs, locations, and types.

Corresponding objects of Buyer, Apartment, and Bungalow are instantiated and added to their respective lists. This process facilitates the initialization of the system with real data, allowing users to engage with the application in a meaningful way from the start.

Displaying Details:
Two methods, displayBuyerDetails() and displayPropertyDetails(), are implemented to print the details of buyers and properties in a structured and formatted manner. These methods enhance the user experience by providing clear and accessible information about available options and buyer status. The use of formatted output ensures that key details are easily readable, facilitating informed decision-making by the buyers as they navigate the property selection process.

Buyer Sorting:
The buyers are sorted by their budgets in descending order, allowing those with higher financial capacity to make purchases first. This sorting mechanism ensures a fair and efficient transaction process, maximizing the likelihood of successful sales for available properties. By prioritizing higher-budget buyers, the application strategically manages property sales, improving the overall efficiency of the system and enhancing the user experience.

Purchasing Loop:
The purchasing loop is the core operational part of the application, allowing each buyer to interact with the property market. For every buyer, the program prompts them to specify a property type (either Apartment or Bungalow). It subsequently displays the suitable properties that match the

buyer’s specified type and budget. Buyers are then able to select a property by entering its ID.
The program checks whether the selected property exists and if the buyer can afford it. It incorporates robust error handling, throwing exceptions as needed to inform the buyer if a selected property is already sold or exceeds their budget. This meticulous handling of the purchasing logic ensures that all transactions are valid, preserving data integrity and enhancing user trust in the system.

Transaction Summary:
After all transactions are completed, the printTransactionSummary() method generates a summary report for all buyers. This report includes details such as the name of each buyer, the properties they purchased, the type of properties, the total amount spent, and their remaining budget. This summary not only serves as a confirmation for the buyers but also provides a comprehensive overview of the transactions conducted during the session. The ability to review transaction outcomes enhances transparency and fosters a positive user experience, encouraging buyers to return for future transactions.

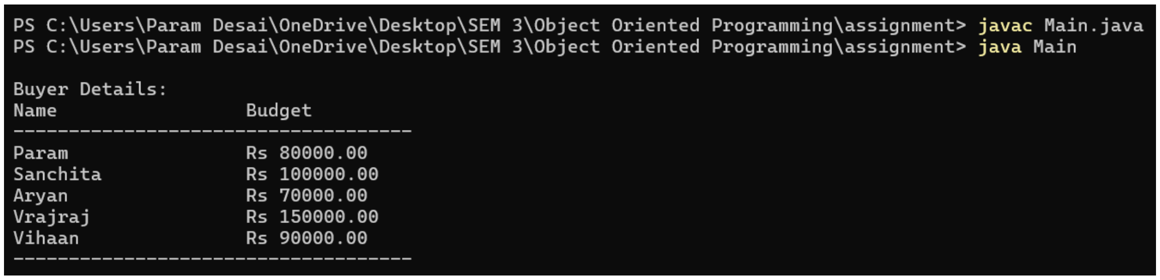
V. Output Formatting and User Interaction

The output of the property management system provides a comprehensive and user-friendly summary of the interactions between buyers and properties. Initially, buyers' details, including names and budgets, are displayed, offering transparency into their purchasing capabilities.

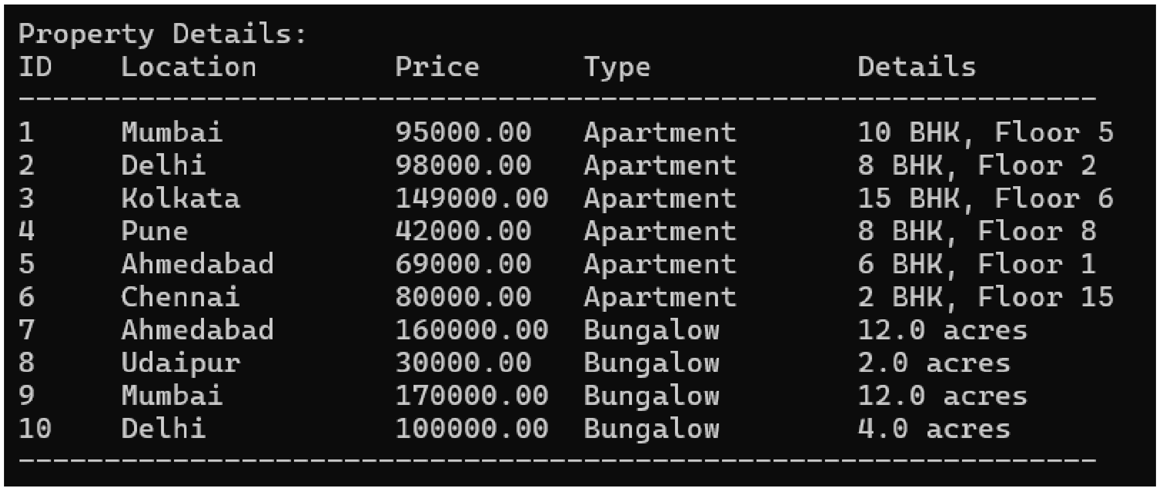
Following this, the properties available for sale are listed with detailed information, including location, price, and type, allowing buyers to make informed choices. As each buyer proceeds to select a property, the system provides feedback regarding the availability and affordability of the chosen properties, ensuring a smooth transaction process.

Upon completion of the buying operations, a transaction summary is generated, detailing each buyer's purchases, the amount spent, and their remaining budget. This summary not only confirms successful transactions but also highlights the efficiency of the system in managing multiple buyers and properties simultaneously.

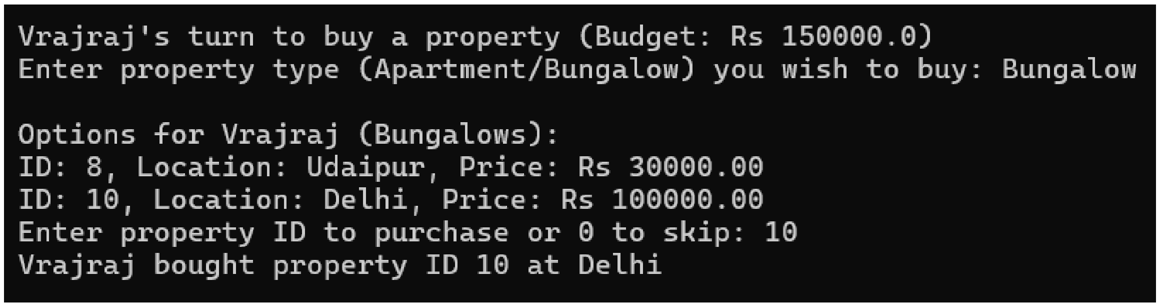
VI. Outputs



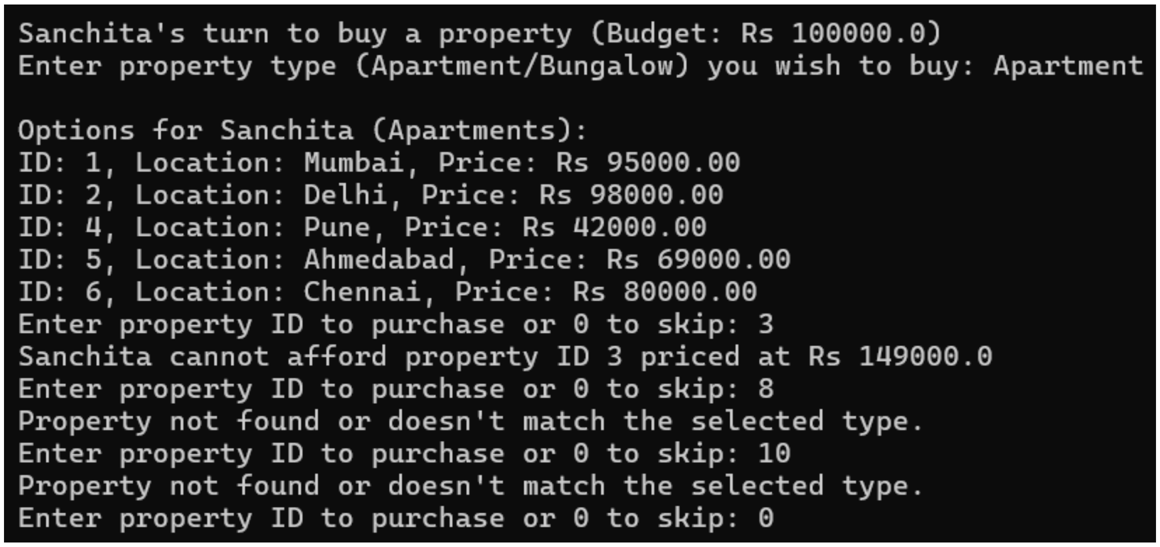
```
PS C:\Users\Param Desai\OneDrive\Desktop\SEM 3\Object Oriented Programming\assignment> javac Main.java
PS C:\Users\Param Desai\OneDrive\Desktop\SEM 3\Object Oriented Programming\assignment> java Main

Buyer Details:
Name                 Budget
------------------------------------
Param                Rs 80000.00
Sanchita             Rs 100000.00
Aryan                Rs 70000.00
Vrajraj              Rs 150000.00
Vihaan               Rs 90000.00
------------------------------------
```



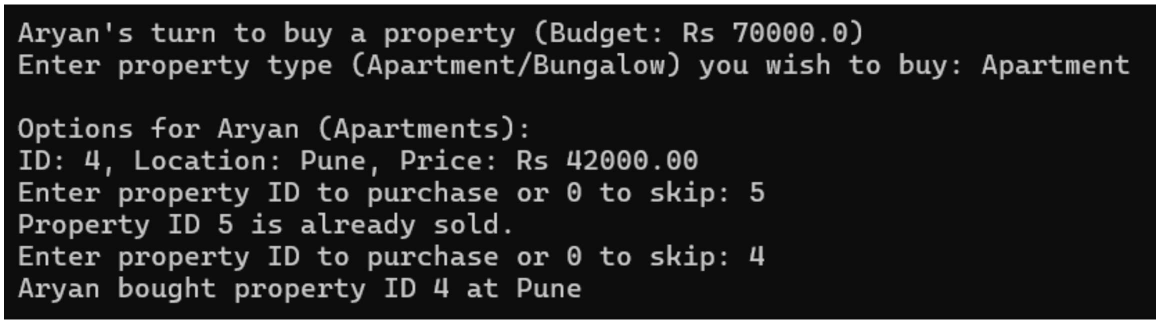
```
Property Details:
ID    Location        Price      Type            Details
---------------------------------------------------------------
1     Mumbai          95000.00   Apartment       10 BHK, Floor 5
2     Delhi           98000.00   Apartment       8 BHK, Floor 2
3     Kolkata         149000.00  Apartment       15 BHK, Floor 6
4     Pune            42000.00   Apartment       8 BHK, Floor 8
5     Ahmedabad       69000.00   Apartment       6 BHK, Floor 1
6     Chennai         80000.00   Apartment       2 BHK, Floor 15
7     Ahmedabad       160000.00  Bungalow        12.0 acres
8     Udaipur         30000.00   Bungalow        2.0 acres
9     Mumbai          170000.00  Bungalow        12.0 acres
10    Delhi           100000.00  Bungalow        4.0 acres
---------------------------------------------------------------
```



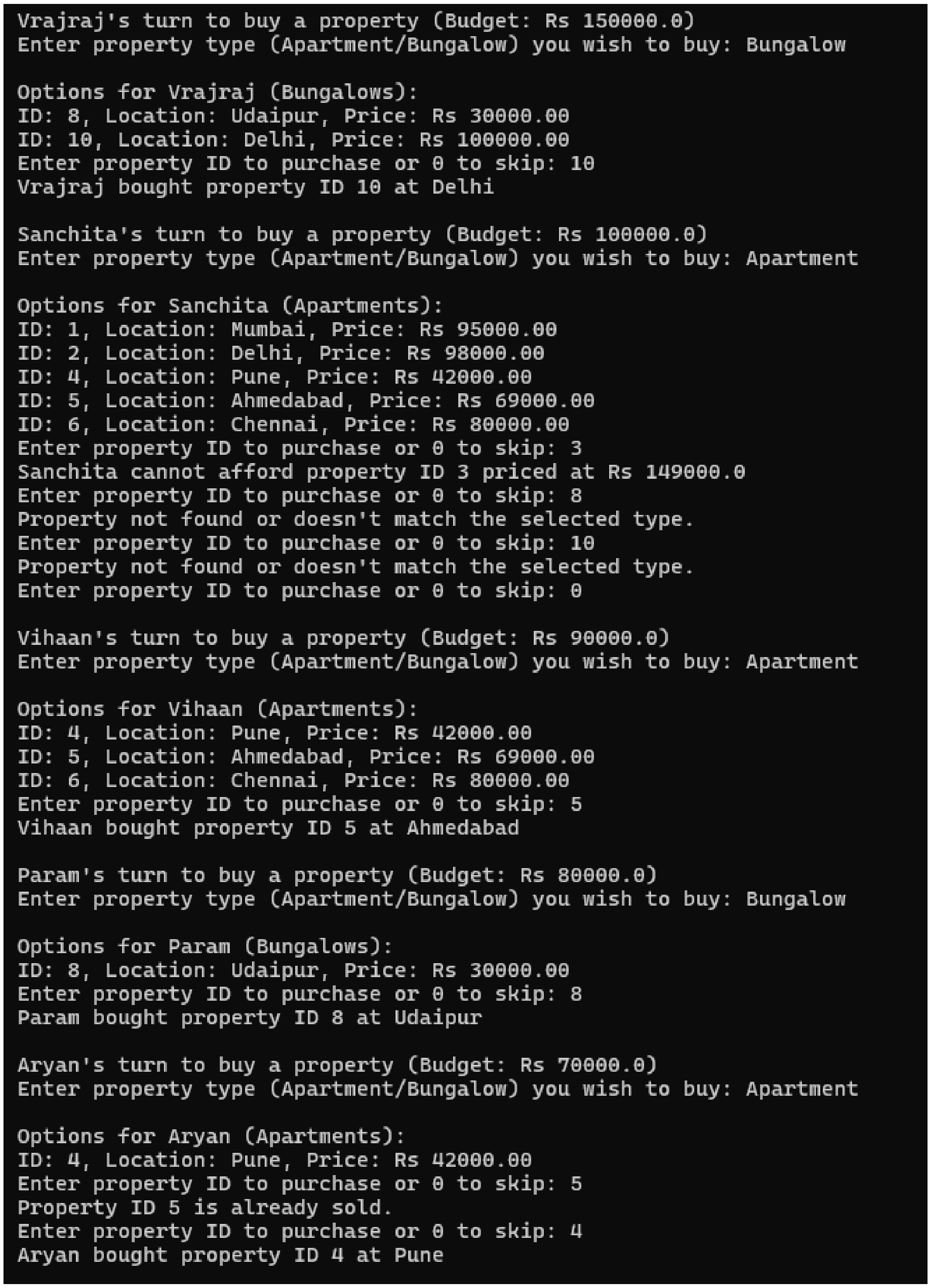
```
Vrajraj's turn to buy a property (Budget: Rs 150000.0)
Enter property type (Apartment/Bungalow) you wish to buy: Bungalow

Options for Vrajraj (Bungalows):
ID: 8, Location: Udaipur, Price: Rs 30000.00
ID: 10, Location: Delhi, Price: Rs 100000.00
Enter property ID to purchase or 0 to skip: 10
Vrajraj bought property ID 10 at Delhi
```



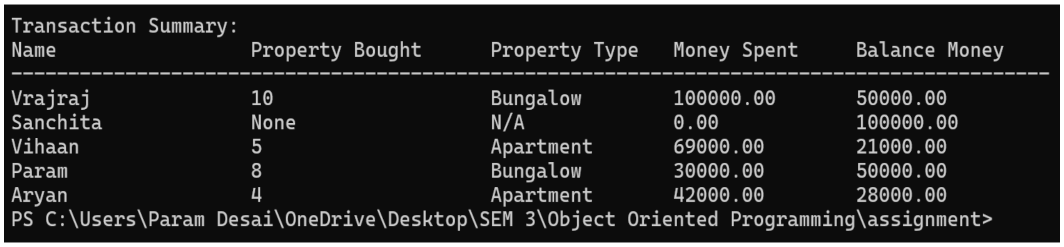
```
Sanchita's turn to buy a property (Budget: Rs 100000.0)
Enter property type (Apartment/Bungalow) you wish to buy: Apartment

Options for Sanchita (Apartments):
ID: 1, Location: Mumbai, Price: Rs 95000.00
ID: 2, Location: Delhi, Price: Rs 98000.00
ID: 4, Location: Pune, Price: Rs 42000.00
ID: 5, Location: Ahmedabad, Price: Rs 69000.00
ID: 6, Location: Chennai, Price: Rs 80000.00
Enter property ID to purchase or 0 to skip: 3
Sanchita cannot afford property ID 3 priced at Rs 149000.0
Enter property ID to purchase or 0 to skip: 8
Property not found or doesn't match the selected type.
Enter property ID to purchase or 0 to skip: 10
Property not found or doesn't match the selected type.
Enter property ID to purchase or 0 to skip: 0
```



```
Aryan's turn to buy a property (Budget: Rs 70000.0)
Enter property type (Apartment/Bungalow) you wish to buy: Apartment

Options for Aryan (Apartments):
ID: 4, Location: Pune, Price: Rs 42000.00
Enter property ID to purchase or 0 to skip: 5
Property ID 5 is already sold.
Enter property ID to purchase or 0 to skip: 4
Aryan bought property ID 4 at Pune
```



```
Vrajraj's turn to buy a property (Budget: Rs 150000.0)
Enter property type (Apartment/Bungalow) you wish to buy: Bungalow

Options for Vrajraj (Bungalows):
ID: 8, Location: Udaipur, Price: Rs 30000.00
ID: 10, Location: Delhi, Price: Rs 100000.00
Enter property ID to purchase or 0 to skip: 10
Vrajraj bought property ID 10 at Delhi

Sanchita's turn to buy a property (Budget: Rs 100000.0)
Enter property type (Apartment/Bungalow) you wish to buy: Apartment

Options for Sanchita (Apartments):
ID: 1, Location: Mumbai, Price: Rs 95000.00
ID: 2, Location: Delhi, Price: Rs 98000.00
ID: 4, Location: Pune, Price: Rs 42000.00
ID: 5, Location: Ahmedabad, Price: Rs 69000.00
ID: 6, Location: Chennai, Price: Rs 80000.00
Enter property ID to purchase or 0 to skip: 3
Sanchita cannot afford property ID 3 priced at Rs 149000.0
Enter property ID to purchase or 0 to skip: 8
Property not found or doesn't match the selected type.
Enter property ID to purchase or 0 to skip: 10
Property not found or doesn't match the selected type.
Enter property ID to purchase or 0 to skip: 0

Vihaan's turn to buy a property (Budget: Rs 90000.0)
Enter property type (Apartment/Bungalow) you wish to buy: Apartment

Options for Vihaan (Apartments):
ID: 4, Location: Pune, Price: Rs 42000.00
ID: 5, Location: Ahmedabad, Price: Rs 69000.00
ID: 6, Location: Chennai, Price: Rs 80000.00
Enter property ID to purchase or 0 to skip: 5
Vihaan bought property ID 5 at Ahmedabad

Param's turn to buy a property (Budget: Rs 80000.0)
Enter property type (Apartment/Bungalow) you wish to buy: Bungalow

Options for Param (Bungalows):
ID: 8, Location: Udaipur, Price: Rs 30000.00
Enter property ID to purchase or 0 to skip: 8
Param bought property ID 8 at Udaipur

Aryan's turn to buy a property (Budget: Rs 70000.0)
Enter property type (Apartment/Bungalow) you wish to buy: Apartment

Options for Aryan (Apartments):
ID: 4, Location: Pune, Price: Rs 42000.00
Enter property ID to purchase or 0 to skip: 5
Property ID 5 is already sold.
Enter property ID to purchase or 0 to skip: 4
Aryan bought property ID 4 at Pune
```



```
Transaction Summary:
Name                Property Bought     Property Type   Money Spent     Balance Money
------------------------------------------------------------------------------------------
Vrajraj             10                  Bungalow        100000.00       50000.00
Sanchita            None                N/A             0.00            100000.00
Vihaan              5                   Apartment       69000.00        21000.00
Param               8                   Bungalow        30000.00        50000.00
Aryan               4                   Apartment       42000.00        28000.00
PS C:\Users\Param Desai\OneDrive\Desktop\SEM 3\Object Oriented Programming\assignment>
```

VII. Conclusion

The Real Estate Management System demonstrates effective usage of Java’s OOP features. By organizing the program around class hierarchies, it shows how real-world systems can be translated into code through abstraction, inheritance, and encapsulation. The project reinforces the flexibility and robustness of OOP in Java, highlighting its potential to create structured and interactive applications