OOP Assignment 3: E-Commerce System

May 2025

1. Key Information

Item	Description
Purpose	This assignment will develop your skills in designing, constructing, testing, and documenting a small Java program according to object-oriented programming principles. This assessment relates to the following learning outcomes: • Applying object-oriented design principles to create reusable and maintainable software • Implementing class hierarchies and interfaces to model real-world systems • Working with data storage and manipulation using Java collections
Your task	This assignment is an individual task where you will write Java code for a simple e-commerce information management system following object-oriented principles as per the specification.
Due Date	2 weeks
Submission	• Via Moodle Assignment Submission. • Git repository check-ins will be used to assess the development history • MOSS will be used for similarity checking of all submissions.
Assessment Criteria	The following aspects will be assessed: 1. Program functionality according to requirements 2. Object-oriented design and adherence to Java coding standards 3. Comprehensive documentation of code
Late Penalties	• 10% deduction per calendar day or part thereof for up to one week • Submissions more than 7 calendar days after the due date will receive a mark of zero (0) and no assessment feedback will be provided.
Support Resources	See Moodle Assessment page and Section 7 in this document

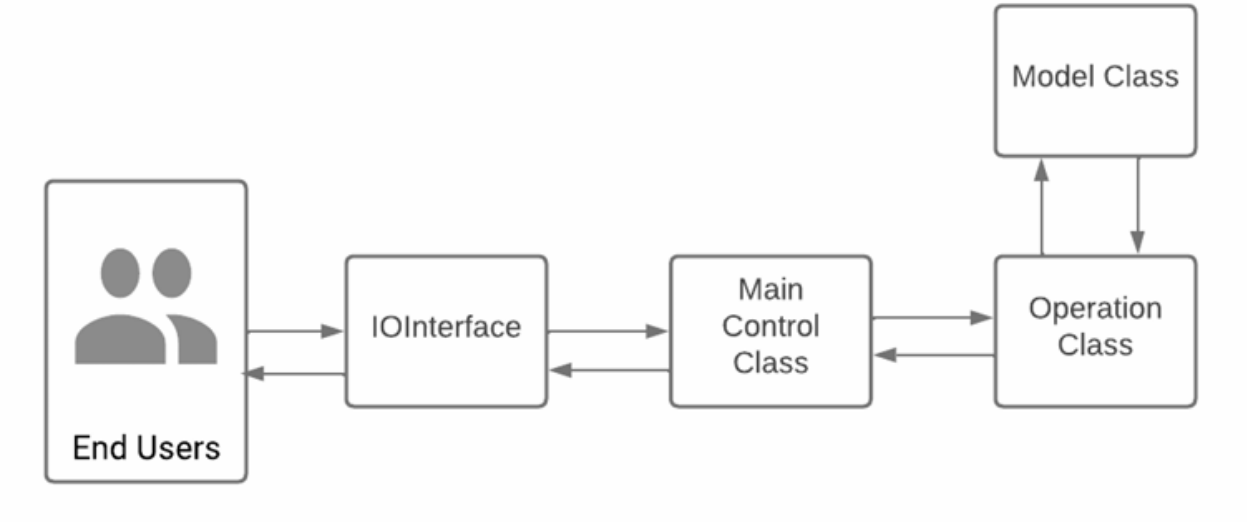
2. Instruction

This assignment requires you to create an e-commerce system which allows customers to log in to the system, perform shopping operations like purchasing products, viewing order history, and showing user consumption reports. Admin users need to be created to manage the whole system, with capabilities to create/delete/view customers, products, and all orders. Besides the management part, admin users can view statistical figures about this system. Since the whole system is executed in the command line interface, you should design a well-formatted user interface and display appropriate messages to guide users.

In this assignment, we are going to implement a system using proper object-oriented principles. As you can see from the diagram below, we have four main components:

1. **Model Classes**: Define the data structure and business logic of the application
2. **Operation Classes**: Handle data access operations using the model classes
3. **Main Control Class**: Handles the main business logic
4. **IOInterface Class**: Handles all user interaction

With this design pattern, **the `System.out.println()` and `Scanner` methods only exist in the I/O interface class. No other classes should have these methods.** The file reading/writing operations happen in the operation classes, which simulate database activities.



System Architecture

All the Operation classes should follow the Singleton pattern, with private constructors and static getInstance() methods. When you use these methods, you should obtain the single instance and use it to invoke methods.

2.1. User Class

User is the abstract base class for Customer and Admin classes.

Required Class Variables

- userId (String): User's unique identifier (format: u_10 digits)
- userName (String): User's name
- userPassword (String): User's password (should be encrypted for storage)
- userRegisterTime (String): User's registration time
- userRole (String): User's role ("customer" or "admin")

Required Methods

2.1.1. Constructors

```
/**
 * Constructs a user object.
 * @param userId Must be unique, format: u_10 digits, such as u_1234567890
 * @param userName The user's name
 * @param userPassword The user's password
 * @param userRegisterTime Format: "DD-MM-YYYY_HH:MM:SS"
 * @param userRole Default value: "customer"
 */
public User(String userId, String userName, String userPassword,
            String userRegisterTime, String userRole) {
    // Implementation
}

/**
 * Default constructor
 */
public User() {
    // Implementation with default values
}
```

2.1.2. toString()

```
/**
 * Returns the user Information as a formatted string.
 * @return String in JSON-like format
 */
@Override
public String toString() {
    // Return in format: {"user_id":"u_1234567890", "user_name":"xxx",
    // "user_password":"xxx", "user_register_time":"xxx",
"user_role":"customer"}
}
```

Note: - All the users are saved in the file "data/users.txt".

2.2. Customer Class

Customer class extends the User class.

Required Class Variables

- userEmail (String): Customer's email address
- userMobile (String): Customer's mobile phone number

Required Methods

2.2.1. Constructors

```
/**
 * Constructs a customer object.
 * @param userId Must be unique, format: u_10 digits, such as u_1234567890
 * @param userName The user's name
 * @param userPassword The user's password
 * @param userRegisterTime Format: "DD-MM-YYYY_HH:MM:SS"
 * @param userRole Default value: "customer"
 * @param userEmail The customer's email address
 * @param userMobile The customer's mobile number
 */
public Customer(String userId, String userName, String userPassword,
                String userRegisterTime, String userRole,
                String userEmail, String userMobile) {
    // Implementation
}

/**
 * Default constructor
 */
public Customer() {
    // Implementation with default values
}
```

2.2.2. toString()

```
/**
 * Returns the customer information as a formatted string.
 * @return String in JSON-like format
 */
@Override
public String toString() {
    // Return in format: {"user_id":"u_1234567890", "user_name":"xxx",
    // "user_password":"xxx", "user_register_time":"xxx",
"user_role":"customer",
    // "user_email":"xxx@gmail.com", "user_mobile":"0412345689"}
}
```

Note: - All the customers are saved in the file "data/users.txt".

2.3. Admin Class

Admin class extends the User class.

Required Class Variables

- None (You can add if you need)

Required Methods

2.3.1. Constructors

```
/**
 * Constructs an admin object.
 * @param userId Must be unique, format: u_10 digits, such as u_1234567890
 * @param userName The user's name
 * @param userPassword The user's password
 * @param userRegisterTime Format: "DD-MM-YYYY_HH:MM:SS"
 * @param userRole Default value: "admin"
 */
public Admin(String userId, String userName, String userPassword,
             String userRegisterTime, String userRole) {
    // Implementation
}

/**
 * Default constructor
 */
public Admin() {
    // Implementation with default values
}
```

2.3.2. toString()

```
/**
 * Returns the admin's attributes as a formatted string.
 * @return String in JSON-like format
 */
@Override
public String toString() {
    // Return in format: {"user_id":"u_1234567890", "user_name":"xxx",
    // "user_password":"xxx", "user_register_time":"xxx",
"user_role":"admin"}
}
```

Note: - All the admins are saved in the file “data/users.txt”.

2.4. Product Class

Model class of product.

Required Class Variables

- proId (String): Product's unique identifier
- proModel (String): Product model
- proCategory (String): Product category
- proName (String): Product name
- proCurrentPrice (double): Current price of the product
- proRawPrice (double): Original price of the product
- proDiscount (double): Discount percentage
- proLikesCount (int): Number of likes

Required Methods

2.4.1. Constructors

```
/**
 * Constructs a product object.
 * @param proId Product ID (must be unique)
 * @param proModel Product model
 * @param proCategory Product category
 * @param proName Product name
 * @param proCurrentPrice Current price of the product
 * @param proRawPrice Original price of the product
 * @param proDiscount Discount percentage
 * @param proLikesCount Number of likes
 */
public Product(String proId, String proModel, String proCategory,
               String proName, double proCurrentPrice, double proRawPrice,
               double proDiscount, int proLikesCount) {
    // Implementation
}

/**
 * Default constructor
 */
public Product() {
    // Implementation with default values
}
```

2.4.2. toString()

```
/**
 * Returns the product information as a formatted string.
 * @return String in JSON-like format
 */
@Override
public String toString() {
    // Return in format: {"pro_id":"xxx", "pro_model":"xxx",
"pro_category":"xxx",
    // "pro_name":"xxx", "pro_current_price":"xxx", "pro_raw_price":"xxx",
```

```
    // "pro_discount":"xxx", "pro_likes_count":"xxx"}
}
```

Note: - All the products are saved in the files “data/products.txt”.

2.5. Order Class

Model class of order.

Required Class Variables

- orderId (String): Order’s unique identifier (format: o_5 digits)
- userId (String): ID of the user who placed the order
- proId (String): ID of the product ordered
- orderTime (String): Time when the order was placed (format: “DD-MM-YYYY_HH:MM:SS”)

Required Methods

2.5.1. Constructors

```
/**
 * Constructs an order object.
 * @param orderId Must be a unique string, format is o_5 digits such as
o_12345
 * @param userId ID of the user who placed the order
 * @param proId ID of the product ordered
 * @param orderTime Format: "DD-MM-YYYY_HH:MM:SS"
 */
public Order(String orderId, String userId, String proId, String orderTime) {
    // Implementation
}

/**
 * Default constructor
 */
public Order() {
    // Implementation with default values
}
```

2.5.2. toString()

```
/**
 * Returns the order information as a formatted string.
 * @return String in JSON-like format
 */
@Override
public String toString() {
    // Return in format: {"order_id":"xxx", "user_id":"xxx", "pro_id":"xxx",
    // "order_time":"xxx"}
}
```

Note:
- All the orders are saved in the file “data/orders.txt”.
- To reduce the program difficulty, we assume each order only has one product.

2.6. UserOperation Class

Contains all the operations related to a user. This class follows the Singleton pattern.

Required Class Variables

- None (You can add if you need)

Required Methods

2.6.1. getInstance()

```
/**
 * Returns the single instance of UserOperation.
 * @return UserOperation instance
 */
public static UserOperation getInstance() {
    // Implementation
}
```

2.6.2. generateUniqueUserId()

```
/**
 * Generates and returns a 10-digit unique user id starting with 'u_'
 * every time when a new user is registered.
 * @return A string value in the format 'u_10digits', e.g., 'u_1234567890'
 */
public String generateUniqueUserId() {
    // Implementation
}
```

2.6.3. encryptPassword()

```
/**
 * Encode a user-provided password.
 * Encryption steps:
 * 1. Generate a random string with a length equal to two times
 *    the length of the user-provided password. The random string
 *    should consist of characters chosen from a-zA-Z0-9.
 * 2. Combine the random string and the input password text to
 *    create an encrypted password, following the rule of selecting
 *    two letters sequentially from the random string and
 *    appending one letter from the input password. Repeat until all
 *    characters in the password are encrypted. Finally, add "^^" at
 *    the beginning and "$$" at the end of the encrypted password.
 *
 * @param userPassword The password to encrypt
 * @return Encrypted password
 */
```

```
public String encryptPassword(String userPassword) {
    // Implementation
}
```

2.6.4. decryptPassword()

```
/**
 * Decode the encrypted password with a similar rule as the encryption
method.
 * @param encryptedPassword The encrypted password to decrypt
 * @return Original user-provided password
 */
public String decryptPassword(String encryptedPassword) {
    // Implementation
}
```

2.6.5. checkUsernameExist()

```
/**
 * Verify whether a user is already registered or exists in the system.
 * @param userName The username to check
 * @return true if exists, false otherwise
 */
public boolean checkUsernameExist(String userName) {
    // Implementation
}
```

2.6.6. validateUsername()

```
/**
 * Validate the user's name. The name should only contain letters or
 * underscores, and its length should be at least 5 characters.
 * @param userName The username to validate
 * @return true if valid, false otherwise
 */
public boolean validateUsername(String userName) {
    // Implementation
}
```

2.6.7. validatePassword()

```
/**
 * Validate the user's password. The password should contain at least
 * one letter (uppercase or lowercase) and one number. The length
 * must be greater than or equal to 5 characters.
 * @param userPassword The password to validate
 * @return true if valid, false otherwise
 */
public boolean validatePassword(String userPassword) {
    // Implementation
}
```

2.6.8. login()

```
/**
 * Verify the provided user's name and password combination against
 * stored user data to determine the authorization status.
 * @param userName The username for login
 * @param userPassword The password for login
 * @return A User object (Customer or Admin) if successful, null otherwise
 */
public User login(String userName, String userPassword) {
    // Implementation
}
```

2.7. CustomerOperation Class

Contains all the operations related to the customer. This class follows the Singleton pattern.

Required Class Variables

- None

Required Methods

2.7.1. getInstance()

```
/**
 * Returns the single instance of CustomerOperation.
 * @return CustomerOperation instance
 */
public static CustomerOperation getInstance() {
    // Implementation
}
```

2.7.2. validateEmail()

```
/**
 * Validate the provided email address format. An email address
 * consists of username@domain.extension format.
 * @param userEmail The email to validate
 * @return true if valid, false otherwise
 */
public boolean validateEmail(String userEmail) {
    // Implementation
}
```

2.7.3. validateMobile()

```
/**
 * Validate the provided mobile number format. The mobile number
 * should be exactly 10 digits long, consisting only of numbers,
 * and starting with either '04' or '03'.
 * @param userMobile The mobile number to validate
 * @return true if valid, false otherwise
```

```
 */
public boolean validateMobile(String userMobile) {
    // Implementation
}
```

2.7.4. registerCustomer()

```
/**
 * Save the information of the new customer into the data/users.txt file.
 * @param userName Customer's username
 * @param userPassword Customer's password
 * @param userEmail Customer's email
 * @param userMobile Customer's mobile number
 * @return true if success, false if failure
 */
public boolean registerCustomer(String userName, String userPassword,
                                String userEmail, String userMobile) {
    // Implementation
}
```

Notes:

- Apply validations in this method to ensure all values are valid.
- If the username exists in the database, return false.
- A unique user id is required when registering a new user.
- Register time should be obtained using the current system time.
- If the user registers successfully, return true and write the customer info into the database. (txt file)

2.7.5. updateProfile()

```
/**
 * Update the given customer object's attribute value. According to
 * different attributes, perform appropriate validations.
 * @param attributeName The attribute to update
 * @param value The new value
 * @param customerObject The customer object to update
 * @return true if updated, false if failed
 */
public boolean updateProfile(String attributeName, String value,
                             Customer customerObject) {
    // Implementation
}
```

2.7.6. deleteCustomer()

```
/**
 * Delete the customer from the data/users.txt file based on the
 * provided customer_id.
 * @param customerId The ID of the customer to delete
 * @return true if deleted, false if failed
 */
public boolean deleteCustomer(String customerId) {
```

```
    // Implementation
}
```

2.7.7. getCustomerList()

```
/**
 * Retrieve one page of customers from the data/users.txt.
 * One page contains a maximum of 10 customers.
 * @param pageNumber The page number to retrieve
 * @return A List of Customer objects, the current page number, and total
pages
 */
public CustomerListResult getCustomerList(int pageNumber) {
    // Implementation
}
```

Note: Create a helper class `CustomerListResult` to hold the list, current page, and total pages.

2.7.8. deleteAllCustomers()

```
/**
 * Removes all the customers from the data/users.txt file.
 */
public void deleteAllCustomers() {
    // Implementation
}
```

2.8. AdminOperation Class

Contains all the operations related to the admin. This class follows the Singleton pattern.

Required Class Variables

- None

Required Methods

2.8.1. getInstance()

```
/**
 * Returns the single instance of AdminOperation.
 * @return AdminOperation instance
 */
public static AdminOperation getInstance() {
    // Implementation
}
```

2.8.2. registerAdmin()

```
/**
 * Creates an admin account. This function should be called when
 * the system starts. The same admin account should not be
 * registered multiple times.
```

```
 */
public void registerAdmin() {
    // Implementation
}
```

2.9. ProductOperation Class

Contains all the operations related to the product. This class follows the Singleton pattern.

Required Class Variables

- None

Required Methods

2.9.1. getInstance()

```
/**
 * Returns the single instance of ProductOperation.
 * @return ProductOperation instance
 */
public static ProductOperation getInstance() {
    // Implementation
}
```

2.9.2. extractProductsFromFiles()

```
/**
 * Extracts product information from the given product data files.
 * The data is saved into the data/products.txt file.
 */
public void extractProductsFromFiles() {
    // Implementation
}
```

2.9.3. getProductList()

```
/**
 * Retrieves one page of products from the database.
 * One page contains a maximum of 10 items.
 * @param pageNumber The page number to retrieve
 * @return A list of Product objects, current page number, and total pages
 */
public ProductListResult getProductList(int pageNumber) {
    // Implementation
}
```

Note: Create a helper class `ProductListResult` to hold the list, current page, and total pages.

2.9.4. deleteProduct()

```
/**
 * Deletes the product from the system based on the provided product_id.
```

```
 * @param productId The ID of the product to delete
 * @return true if successful, false otherwise
 */
public boolean deleteProduct(String productId) {
    // Implementation
}
```

2.9.5. getProductListByKeyword()

```
/**
 * Retrieves all products whose name contains the keyword (case insensitive).
 * @param keyword The search keyword
 * @return A list of Product objects matching the keyword
 */
public List<Product> getProductListByKeyword(String keyword) {
    // Implementation
}
```

2.9.6. getProductById()

```
/**
 * Returns one product object based on the given product_id.
 * @param productId The ID of the product to retrieve
 * @return A Product object or null if not found
 */
public Product getProductById(String productId) {
    // Implementation
}
```

2.9.7. generateCategoryFigure()

```
/**
 * Generates a bar chart showing the total number of products
 * for each category in descending order.
 * Saves the figure into the data/figure folder.
 */
public void generateCategoryFigure() {
    // Implementation using Java charting library
}
```

2.9.8. generateDiscountFigure()

```
/**
 * Generates a pie chart showing the proportion of products that have
 * a discount value less than 30, between 30 and 60 inclusive,
 * and greater than 60.
 * Saves the figure into the data/figure folder.
 */
public void generateDiscountFigure() {
    // Implementation using Java charting library
}
```

2.9.9. generateLikesCountFigure()

```
/**
 * Generates a chart displaying the sum of products' likes_count
 * for each category in ascending order.
 * Saves the figure into the data/figure folder.
 */
public void generateLikesCountFigure() {
    // Implementation using Java charting library
}
```

2.9.10. generateDiscountLikesCountFigure()

```
/**
 * Generates a scatter chart showing the relationship between
 * likes_count and discount for all products.
 * Saves the figure into the data/figure folder.
 */
public void generateDiscountLikesCountFigure() {
    // Implementation using Java charting library
}
```

2.9.11. deleteAllProducts()

```
/**
 * Removes all product data in the data/products.txt file.
 */
public void deleteAllProducts() {
    // Implementation
}
```

2.10. OrderOperation Class

Contains all the operations related to the order. This class follows the Singleton pattern.

Required Class Variables

- None

Required Methods

2.10.1. getInstance()

```
/**
 * Returns the single instance of OrderOperation.
 * @return OrderOperation instance
 */
public static OrderOperation getInstance() {
    // Implementation
}
```

2.10.2. generateUniqueOrderId()

```
/**
 * Generates and returns a 5-digit unique order id starting with "o_".
```

```
 * @return A string such as o_12345
 */
public String generateUniqueOrderId() {
    // Implementation
}
```

2.10.3. createAnOrder()

```
/**
 * Creates a new order with a unique order id and saves it to the
 * data/orders.txt file.
 * @param customerId The ID of the customer making the order
 * @param productId The ID of the product being ordered
 * @param createTime The time of order creation (null for current time)
 * @return true if successful, false otherwise
 */
public boolean createAnOrder(String customerId, String productId,
                             String createTime) {
    // Implementation
}
```

2.10.4. deleteOrder()

```
/**
 * Deletes the order from the data/orders.txt file based on order_id.
 * @param orderId The ID of the order to delete
 * @return true if successful, false otherwise
 */
public boolean deleteOrder(String orderId) {
    // Implementation
}
```

2.10.5. getOrderList()

```
/**
 * Retrieves one page of orders from the database belonging to the
 * given customer. One page contains a maximum of 10 items.
 * @param customerId The ID of the customer
 * @param pageNumber The page number to retrieve
 * @return A list of Order objects, current page number, and total pages
 */
public OrderListResult getOrderList(String customerId, int pageNumber) {
    // Implementation
}
```

Note: Create a helper class `OrderListResult` to hold the list, current page, and total pages.

2.10.6. generateTestOrderData()

```
/**
 * Automatically generates test data including customers and orders.
 * Creates 10 customers and randomly generates 50-200 orders for each.
 * Order times should be scattered across different months of the year.
```

```
 */
public void generateTestOrderData() {
    // Implementation
}
```

2.10.7. generateSingleCustomerConsumptionFigure()

```
/**
 * Generates a chart showing the consumption (sum of order prices)
 * across 12 different months for the given customer.
 * @param customerId The ID of the customer
 */
public void generateSingleCustomerConsumptionFigure(String customerId) {
    // Implementation using Java charting library
}
```

2.10.8. generateAllCustomersConsumptionFigure()

```
/**
 * Generates a chart showing the consumption (sum of order prices)
 * across 12 different months for all customers.
 */
public void generateAllCustomersConsumptionFigure() {
    // Implementation using Java charting library
}
```

2.10.9. generateAllTop10BestSellersFigure()

```
/**
 * Generates a graph showing the top 10 best-selling products
 * sorted in descending order.
 */
public void generateAllTop10BestSellersFigure() {
    // Implementation using Java charting library
}
```

2.10.10. deleteAllOrders()

```
/**
 * Removes all data in the data/orders.txt file.
 */
public void deleteAllOrders() {
    // Implementation
}
```

2.11. IOInterface Class

This Class handles all the I/O operations. All `System.out.println()` and `Scanner` operations should be defined in this class.

Required Class Variables

- None

Required Methods

2.11.1. getInstance()

```
/**
 * Returns the single instance of IOInterface.
 * @return IOInterface instance
 */
public static IOInterface getInstance() {
    // Implementation
}
```

2.11.2. getUserInput()

```
/**
 * Accept user input.
 * @param message The message to display for input prompt
 * @param numOfArgs The number of arguments expected
 * @return An array of strings containing the arguments
 */
public String[] getUserInput(String message, int numOfArgs) {
    // Implementation
}
```

Notes:

- The message is used for the input prompt.
- User inputs have only one format with all arguments connected by a whitespace ” “.
- If users input more than numOfArgs arguments, ignore the extras.
- If users input fewer than numOfArgs arguments, fill the remaining with empty strings.

2.11.3. mainMenu()

```
/**
 * Display the login menu with options: (1) Login, (2) Register, (3) Quit.
 * The admin account cannot be registered.
 */
public void mainMenu() {
    // Implementation
}
```

2.11.4. adminMenu()

```
/**
 * Display the admin menu with options:
 * (1) Show products
 * (2) Add customers
 * (3) Show customers
 * (4) Show orders
 * (5) Generate test data
 * (6) Generate all statistical figures
 * (7) Delete all data
 * (8) Logout
 */
```

```
public void adminMenu() {
    // Implementation
}
```

2.11.5. customerMenu()

```
/**
 * Display the customer menu with options:
 * (1) Show profile
 * (2) Update profile
 * (3) Show products (user input could be "3 keyword" or "3")
 * (4) Show history orders
 * (5) Generate all consumption figures
 * (6) Logout
 */
public void customerMenu() {
    // Implementation
}
```

2.11.6. showList()

```
/**
 * Prints out different types of lists (Customer, Product, Order).
 * Shows row number, page number, and total page number.
 * @param userRole The role of the current user
 * @param listType The type of list to display
 * @param objectList The list of objects to display
 * @param pageNumber The current page number
 * @param totalPages The total number of pages
 */
public void showList(String userRole, String listType, List<?> objectList,
                     int pageNumber, int totalPages) {
    // Implementation
}
```

2.11.7. printErrorMessage()

```
/**
 * Prints out an error message and shows where the error occurred.
 * @param errorSource The source of the error
 * @param errorMessage The error message
 */
public void printErrorMessage(String errorSource, String errorMessage) {
    // Implementation
}
```

2.11.8. printMessage()

```
/**
 * Print out the given message.
 * @param message The message to print
 */
public void printMessage(String message) {
```

```
    // Implementation
}
```

2.11.9. printObject()

```
/**
 * Print out the object using the toString() method.
 * @param targetObject The object to print
 */
public void printObject(Object targetObject) {
    // Implementation
}
```

2.12. Main File

In this file, you will construct the main control logic for the application (e.g., main() method). The design and implementation is up to you but must include the menu items outlined in sections 2.11.3, 2.11.4, and 2.11.5 using the classes and methods implemented.

You must ensure that your menu and control logic handles exceptions appropriately.

You can break down your code into several methods if you wish, but you need to call any extra-defined methods from the main method. Your instructor will only run your Main.java file.

For each operation that the user performs, try to give enough instructional messages.

For all the tasks above, you can change the class/method/variable names to follow your own naming conventions. It is allowed to add more class variables and methods. However, you need to make sure all the required methods are implemented. Any unused code in your application will receive mark penalties.

2.13. User Manual

It is required to provide user instructions saved into a file named `userManual_{studentid}.pdf` which describes how to use your application. In your PDF, list all the commands used to reach the tasks listed in sections 2.11.3, 2.11.4, and 2.11.5. Your marker will follow your manual to test all the functions. Make sure you demonstrate special cases like validation failures.

Please do not show too much content in this document. No more than 5 pages.

3. Do and Do NOT

Do	Do NOT
• Maintain appropriate citing and referencing	• Leave your assignment in draft mode
• Get support early from the teaching team	• Submit late (10% daily penalty applies)
• Apply for special consideration or extensions	• Attempt to submit after 7 days of the

Do	Do NOT
early if needed	due date

3.1. Important Notes:

- DO NOT use absolute paths. All path issues that cause program crashes will lead to no mark for functionality.
- You must implement all required methods but you may add additional methods if required.
- Ensure file operations include proper exception handling. Any character encoding issues will cause serious mark deduction.
- If one method doesn't work and prevents the program from running, later functionalities will get no mark.
- If any exceptions occur when running your program, you will lose 50% of the allocated function logic marks.
- Add appropriate validation and output messages to make your code user-friendly.
- The assignment must be implemented using Java SE 17 or higher.
- The Java code must follow standard Java conventions and coding style.
- Only use the following libraries: java.util.*, java.io.*, java.time.*, org.json.*, javafx.* (for charts). Use of any other external libraries must be approved.
- Add comprehensive comments to your code, including Javadoc comments for classes and methods.
- This assignment requires ongoing work over several weeks - start early and make progress incrementally.
- Keep up to date with the Backboard where clarifications may be posted.
- Please be careful not to publicly post anything which includes your reasoning, logic, or any part of your work to forums - doing so violates plagiarism/collusion rules.
- In this assessment, you must **NOT use generative artificial intelligence (AI)** to generate any materials or content in relation to the assignment task.

4. Submission Requirements

The following files are to be submitted and must exist in your Git repository:

- All Java source files (*.java)

The above files must be compressed to a .zip file named ass3_{studentid}.zip and submitted via BB.

The Java files must also have been pushed to your Git repository with an appropriate development history. Please ensure your commit messages are meaningful. You only need to push the final version of the PDF file, not its history. DO NOT push the .zip file.

Important submission notes:

- No submissions will be accepted via email.

- Please note we cannot mark any work on the Git Server; you need to ensure that you submit correctly via BB.

- It is your responsibility to ENSURE that the submitted files are the correct files. We strongly recommend downloading your submission prior to finalizing it to verify its contents.

- Please submit several hours before the deadline to avoid last-minute technical issues.

- Marks will be deducted for any of these requirements that are not strictly complied with.