Advanced Java Concepts Report

part of course ITSE311 Syllabus

Object Oriented Programming, Threading and Concurrency, DataBase Connectivity, GUI and Server side

Sanad AlArousi
218 180 1442

1. Introduction

This report outlines and showcases several concepts in programming, which are used to teach students and learners from all background some advanced concepts and understanding of programming, all from the perspective of Java Language.

Java is machine independent thanks to its Java Virtual Machine – JVM -, which takes the compiled java file named Bytecode and runs it itself, unlike other languages that will need a container or Docker technology to make it portable.

Java is also heavily reliant on the concept of Objects, all files are either classes or objects, most functions can be turned into an Object, even data types can be made an Object, allowing that int variable which will turn into Int, to have several methods that it can access for various utilization of that variable. This also implies on Arrays and other kinds of “data structures”.

Java also has a well thought of standard set of libraries, which are called packages. Mainly learners will use java.util , java.sql, java.time, java.file. With every release of java the number of packages and its internal methods expands giving it huge agile development capabilities, just like python.

In this report I will introduce the core teaching concepts of programming, which will be practiced through java, OOP, GUI, DB connectivity, server side functionality and Concurrency.

2. Object-Oriented Programming in JAVA

programming for the purpose of the object means that we are not here to code, we are here only to develop. And from this main goal the 4 concepts of OOP have been built and well defined.

2.1 Encapsulation:

2.1.1 Internal code of the object or the class are none of our concerns, we only need to know the type of inputs it accepts, and the type of outputs It produces, and a general descriptive naming or documentation, to use it. Hence no need to understand the way a .sort() method is written and we should utilize our time and energy focusing on developing the project for the idea. That is ENCAPSULATION.

2.1.2 **Encapsulation in Java** allows you to reach certain variables *only* through a setValue() and a getValue() methods, called setters and getters, this encapsulates variables and prevents manual editing or public sector reach to those variables, hence it can only be reached within its class and through a predefined functions.

2.1.3 As you have already thought, encapsulation overlaps with the “scope” concepts.

2.2 Inheritance:

2.2.1 less coding more developing is here achieved through blueprints, or Classes. Which then allow us to create objects or other classes from them, decreasing time of code and allowing more time for developing, here we call it **ReUsability**.

2.2.2 **Inheritance in Java** is unique as we have “Sub-Classes” and “Super-Classes”, further along we also have the usage of interfaces and abstract classes. Inheritance in java can happen on multilevel, as in GrandFather Class, Father Class, Sun Class, with a slight twerk called “**Interface**”, making the proper naming to be: GrandFather Class, Father Interface, Sun Class.

2.2.3 Inheritance in Java also brings in the concept of Override. Which allows defining a different version of the method inherited from the father interface or the GrandFather Class.

2.3 **Polymorphism:**

2.3.1 less code more development In this case means, every human has the Human.talk() method, however the class Arabs that inherits Human class, will have a different Arab.talk() method output.

2.3.2 **Polymorphism in Java** focuses on Methods Override and doesn’t allow any operator override, yet there are 2 kinds of Method Polymorphism in Java mentioned next:

2.3.3 **Compile-Time Polymorphism** is simply “**Method Overloading**” meant to allow the compiler to choose which function to call based on the kind of inputted parameters given to it, hence 2 methods could have the same name and yet different parameters.

2.3.4 **Run-Time Polymorphism** is in the other hand "**Method Overriding**" the simple kind of Polymorphism that an inheriting class or object redefines a method to fit its specific needs.

2.4 **Abstraction:**

2.4.1 Abstraction is to control what kind of data the developer sees

2.4.2 **Abstraction in Java** is when we use interfaces and abstract methods. Abstract methods are bodiless functions that allow the inheriting class to define-tune that abstract function to what it benefits, as you might have noticed it is very similar in concept to **Method Overriding**, actually, overriding is compulsory for abstract methods In Java

2.4.3 Abstract methods are very confusing at the begging, however its used in a super class to give a general form of a function for the rest of inheriting classes

3. Connecting to a Database using JAVA

One of the best approaches to learning a language is to learn how to connect and send data to a DB, the processes which we intend to make with our programming language are called CRUD processes, which stands for: Create, Read, Update, Delete Processes. In Java we have an SQL package mentioned earlier called java.sql, which gives us access to the structured query language through java.

3.1 to connect to a DataBase we need to first make a "Connection" :), which is a link to the DB in our code, for this the java.sql package give us DriverManager.

3.2 the **DriverManager** then gives us in return a set of useful methods for our **ConnLink**. Some of them are: **ConnLink.getConnection()**, However since **JDBC API 2.0**, the preferred method of connecting to a DB is through a **DataSource()**.and If we utilize the rest of the .sql package properly we can make a the **CRUD** processes.

3.3 The **JDBC API** is to connect to a **MySQL DB**, however for each DB we need its driver and we need to feed it to the DriverManager, for **SQLite3 we use**: (**sqlite-jdbc**.jar and **slf4j-api**.jar), which must be in the project files or Class Path.

3.4 Sending or receiving data is preferred to be done using a GUI, not through a CLI, in which we have AWT and Swing.

3.5 Regardless of the vast use cases of Java, **PHP** and **Rust** remains the Kings of **back-end** processes.

4.GUI in Java

Graphical User Interface is the preferred method for sending and receiving data from a DB, in which the user doesn't have the tension of working with a CMD and has to bear to feel like a hacker.

4.1 Whereas the ideal path to making an Interface is through HTML, CSS and JavaScript, many languages still have its own Libraries/Packages to make a simple one for testing or prototyping,

4.2 **Java GUI packages** are **Swing** and **AWT**, *Swing* is the preferred choice for many for its **light-weight performance** and **variety in components and feels**.

4.3 **Java Foundation Classes** which contains the java swing, also contain Accessibility API for screen readers and Braille displays, which are legally enforced in many countries in the past few years.

4.4 the JFC also has a 2D API, allowing developers to incorporate high-quality graphics, text and images in applications or applets. Java 2D includes extensive APIs for generating and sending high-quality output to printing devices.

4.5 JFC also takes in consideration **Internationalization,** allowing build applications that can interact with users worldwide in their own languages and cultural conventions.

5. Concurrency in Java

Concurrency means dealing with multiple tasks and processes simultaneously, utilizing multi-core and multi thread processors, by allowing each part of the code to run separately and independently. This improves responsiveness and resource utilization, however we have the challenges of "Thread Safety", "Race Conditions", "Coordination and Communication between tasks/threads".

5.1 To do concurrency through Java, Threads and Runnables must be implemented and inherited from. An example of multithreading is making 2 Thread files containing a loop each, Thread1File.java contains a loop that prints Hi, Thread2File.java contains a loop that prints Hello, once we use them in Main both loops will print together.

5.2 **Runnable Interface** *inheritance* is a *more encouraged approach* as it allows better separation of concerns and flexibility. Whereas for **Thread Clas**s, the java doesn't support multiple inheritance, hence your application can't inherit from the GrandFather called Thread Class.

5.3 Thread Safety is when processes or threads awaits data or results from each other, thereby we need to ensure proper synchronization using "synchronized" blocks or classes from the package 'java.util.concurrent',

5.4 Race Conditions happens when multiple threads try to reach or manipulate variables or databases concurrently. If we don't maintain thread safety and sync, errors of the caliber or 'logical errors' will emerge, making debugging and development harder and more complicated.

5.5 'synchronized' keyword, is what prevents threads from accessing "**Critical Sections**" at the same time. A very similar concept in ITGS302.

5.6 DeadLocks, or what I call them **crash events**. Its when a Thread: 1) holds a resource Exclusively and demands a value from another thread or resource, 2) the other thread is waiting for the resource to preempty to give the initial Thread the value it wants, 3) No PreEmptation of Resources allowed. This is obviously resulting in a mess if not maintained the right way.

5.7 **Approaches to prevent DeadLocks In Java:**

5.7.1 Time out Resource acquisition and awaits to acquisition

5.7.2 Use DeadLock detection algorithms allowing the system to check the status of threads automatically and forcefully ends the circular dependency

5.8 A drawback in Java is that its JVM doesn't have a built-in functionality to handle DeadLocks or prevent them, thereby developers have to implement them manually In their applications.

```java
public class Person implements Displayable {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }

    @Override
    public void displayInfo() {
        System.out.println("Name: " + name);
    }

    public void displayInfo(boolean showAge) {
        System.out.println("Name: " + name);
        if (showAge) {
            System.out.println("Age: " + age);
        }
    }
}
```

Figure 1: 2.1.2 Encapsulation

Figure 2: 2.2 Inheritance

```java
public class Employee extends Person {
    private String employeeId;

    public String getEmployeeId() {
        return employeeId;
    }

    public void setEmployeeId(String employeeId) {
        this.employeeId = employeeId;
    }

    @Override
    public void displayInfo() {
        super.displayInfo();
        System.out.println("Employee ID: " + employeeId);
    }
}
```

```java
public class Person implements Displayable {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age > 0) {
            this.age = age;
        }
    }
```

Figure 4: 2.2.3 Implementation of Interface

Figure 3: 2.2 Father of Employee

```java
public interface Displayable {
    void displayInfo();
}
```

Figure 5: Interface for Person Class

```java

    @Override
    public void displayInfo() {
        System.out.println("Name: " + name);
    }

    public void displayInfo(boolean showAge) {
        System.out.println("Name: " + name);
        if (showAge) {
            System.out.println("Age: " + age);
        }
    }
}
```

Figure 6: 2.3 Polymorphism @Override

Figure 7: 3 DB Connectivity & 4 GUI

```java
import javax.swing.*;
import java.awt.event.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.awt.BorderLayout;
import java.sql.ResultSet;

public class DMLOperationGUI extends JFrame implements ActionListener {

    public static Connection createTableGetConn() {
        Connection connection = null;
        try {
            Class.forName("org.sqlite.JDBC");
            connection = DriverManager.getConnection("jdbc:sqlite:SqliteJavaDB.db");
            System.out.println("\n[DB Created/Connected successfuly]");
        } catch (Exception er) {
            System.out.println("/nError");
            System.err.println(er.getClass().getName() + ":" + er.getMessage());
            System.exit(0);
        }
        return connection;
    }

    private Connection connection;
    private final JButton insertButton;
    private final JButton updateButton;
    private final JButton deleteButton;
    private final JButton selectButton;
```

```
31      private final JTextArea outputTextArea;
32      private final JTextField nameField;
33      private final JTextField priceField;
34      private final JTextField quantityField;
35      private final JTextField idField;
36
37
38      public DMLOperationGUI(Connection connection) throws SQLException {
39          this.connection = connection;
40
41          // Initialize UI components
42
43          outputTextArea = new JTextArea(7, 35);
44
45          nameField = new JTextField(15);
46          priceField = new JTextField(10);
47          quantityField = new JTextField(10);
48          idField = new JTextField(10);
49
50          insertButton = new JButton("Insert");
51          updateButton = new JButton("Update");
52          deleteButton = new JButton("Delete");
53          selectButton = new JButton("Select");
54
55          // Add action listeners
56          insertButton.addActionListener(this);
57          updateButton.addActionListener(this);
58          deleteButton.addActionListener(this);
59          selectButton.addActionListener(this);
60
```

Figure 8: 3 DB Connectivity & 4 GUI

```
61          // Layout components
62          JPanel buttonPanel = new JPanel();
63          buttonPanel.add(insertButton);
64          buttonPanel.add(updateButton);
65          buttonPanel.add(deleteButton);
66          buttonPanel.add(selectButton);
67
68          JPanel inputPanel = new JPanel();
69
70          inputPanel.add(new JLabel("Name:"));
71          inputPanel.add(nameField);
72
73          inputPanel.add(new JLabel("Price:"));
74          inputPanel.add(priceField);
75
76          inputPanel.add(new JLabel("Quantity:"));
77          inputPanel.add(quantityField);
78
79          inputPanel.add(new JLabel("ID (Update/Delete):"));
80          inputPanel.add(idField);
81
82          JPanel outputPanel = new JPanel();
83          outputPanel.add(new JScrollPane(outputTextArea));
84
85          getContentPane().setLayout(new BorderLayout());
86          getContentPane().add(buttonPanel, BorderLayout.NORTH);
87          getContentPane().add(inputPanel, BorderLayout.CENTER);
88          getContentPane().add(outputPanel, BorderLayout.SOUTH);
89
90          // Set window properties
```

Figure 9: 3 DB Connectivity & 4 GUI

```
91          pack();
92          setVisible(true);
93          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
94      }
95
96      @Override
97      public void actionPerformed(ActionEvent e) {
98          try {
99              String action = e.getActionCommand();
100             switch (action) {
101                 case "Insert":
102                     // **Insert product functionality**
103                     String name = nameField.getText();
104                     float price = Float.parseFloat(priceField.getText());
105                     int quantity = Integer.parseInt(quantityField.getText());
106                     insertProduct(connection, name, price, quantity);
107                     outputTextArea.setText("Product Inserted Successfully!");
108                     printTableContent(connection, outputTextArea); // Update table after insert
109                     break;
110                 case "Update":
111                     // **Update product functionality**
112                     name = nameField.getText();
113                     price = Float.parseFloat(priceField.getText());
114                     quantity = Integer.parseInt(quantityField.getText());
115                     int id = Integer.parseInt(idField.getText());
116                     updateProduct(connection, id, name, price, quantity);
117                     outputTextArea.setText("Product Updated Successfully!");
118                     printTableContent(connection, outputTextArea); // Update table after update
119                     break;
120                 case "Delete":
```

Figure 10: 3 DB Connectivity & 4 GUI

```
121                     // **Delete product functionality**
122                     id = Integer.parseInt(idField.getText());
123                     deleteProduct(connection, id);
124                     outputTextArea.setText("Product Deleted Successfully!");
125                     printTableContent(connection, outputTextArea); // Update table after delete
126                     break;
127                 case "Select":
128                     // **Select and display product data functionality**
129                     printTableContent(connection, outputTextArea);
130                     break;
131             }
132             // Clear input fields after successful operations
133             nameField.setText("");
134             priceField.setText("");
135             quantityField.setText("");
136             idField.setText("");
137         } catch (Exception ex) {
138             // Handle
139         }
140     }
141
142     private void insertProduct(Connection connection, String name, float price, int quantity) throws SQLException {
143         // Enable auto-commit to simplify the example (consider using transactions for
144         // real applications)
145         connection.setAutoCommit(true);
146
147         String sql = "INSERT INTO Product (p_name, price, quantity) VALUES ('" + name + "', " + price + ", " + quantity
148                 + ")";
149         Statement statement = connection.createStatement();
150         statement.executeUpdate(sql);
```

Figure 11: 3 DB Connectivity & 4 GUI

Figure 12: 3 DB Connectivity & 4 GUI

```
151
152         // No need to commit as auto-commit is enabled
153     }
154
155     private void updateProduct(Connection connection, int id, String name, float price, int quantity)
156             throws SQLException {
157         // Enable auto-commit to simplify the example (consider using transactions for
158         // real applications)
159         connection.setAutoCommit(true);
160
161         String sql = "UPDATE Product SET p_name='" + name + "', price=" + price + ", quantity=" + quantity
162                 + " WHERE p_id=" + id;
163         Statement statement = connection.createStatement();
164         statement.executeUpdate(sql);
165
166         // No need to commit as auto-commit is enabled
167     }
168
169     private void deleteProduct(Connection connection, int id) throws SQLException {
170
171         // Consider using transactions for real applications to ensure data integrity
172         try (Statement statement = connection.createStatement()) {
173             String sql = "DELETE FROM Product WHERE p_id=" + id;
174             statement.executeUpdate(sql);
175         } catch (SQLException sqle) {
176             // Handle specific SQL exceptions (e.g., foreign key constraint violation)
177             outputTextArea.setText("Error deleting product: " + sqle.getMessage());
178             // Consider checking for specific error codes (e.g., ORA-02292 for foreign key
179             // constraint violation)
180             // and providing more informative messages
181             throw sqle; // Re-throw the exception for caller handling (consider if needed)
182         }
183     }
```

```
184
185     private void printTableContent(Connection connection, JTextArea outputTextArea) throws SQLException {
186         try (Statement statement = connection.createStatement()) {
187             String sql = "SELECT * FROM Product";
188             ResultSet resultSet = statement.executeQuery(sql);
189
190             // Clear the output area before displaying new data
191             outputTextArea.setText("");
192
193             // Iterate through the result set and display product information
194             while (resultSet.next()) {
195                 int productId = resultSet.getInt("p_id");
196                 String productName = resultSet.getString("p_name");
197                 float productPrice = resultSet.getFloat("price");
198                 int productQuantity = resultSet.getInt("quantity");
199
200                 String productInfo = String.format("ID: %d     Name: %s\t Price: %.2f\t Quantity: %d\n", productId,
201                         productName, productPrice, productQuantity);
202                 outputTextArea.append(productInfo);
203             }
204         }
205     }
206
207     public static void main(String args[]) throws SQLException {
208         Connection c = null;
209         c = createTableGetConn();
210         DMLOperationGUI gui = new DMLOperationGUI(c);
211     }
212 }
```

Figure 14: 3 DB Connectivity & 4 GUI