

**NATIONAL UNIVERSITY OF SCIENCES & TECHNOLOGY**

**MILITARY COLLEGE OF SIGNALS**



**DATABASE SYSTEMS**  
**(CS-220)**

**LAB PROJECT DOCUMENTATION**

**Submitted by:**

- 1. AYESHA ISRAR**
- 2. MUHAMMAD AHMAD SULTAN**
- 3. SAMAR QAISER**
- 4. SHANDANA IFTIKHAR**

**COURSE: BESE-28**

**SECTION: C**

**Submitted to: DR. AYESHA NASEER**

***Dated: 30-01-2024***

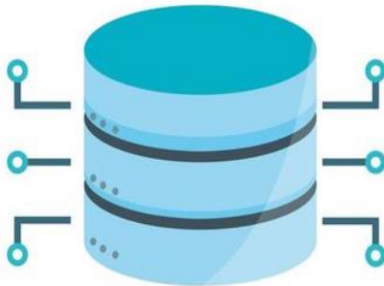
# CATALOG OF THE LAB PROJECT

## Table of Contents

▪ Title of the Lab Project .....	03
▪ Abstract .....	04
▪ History of Blood Bank Management Systems .....	04
▪ Objective .....	05
▪ Scope .....	05
▪ Beyond Current Focus .....	06
▪ Functionality & Implementation .....	07
▪ Entity Relationship Diagram .....	10
▪ MySQL Database Queries .....	13
▪ Java Classes .....	20
▪ Input and Output Specifications .....	31
▪ OOP Principles & Concepts .....	33
▪ Software and Equipment Used .....	34
▪ Output and Execution Flow .....	35
▪ Display .....	37
▪ Future Enhancements .....	44
▪ Potential Questions .....	45
▪ Conclusion .....	46
▪ Reference Citations .....	46

**DEPARTMENT OF COMPUTER SOFTWARE ENGINEERING**  
**Military College of Signals**  
**National University of Sciences and Technology**

[www.mcs.nust.edu.pk](http://www.mcs.nust.edu.pk)



## TITLE OF THE LAB PROJECT

# BLOOD BANK MANAGEMENT SYSTEM

*A Comprehensive Blood Bank Management System in Java, integrating Data Structures & Algorithms, Object-Oriented Programming, and Database Management with MySQL, ensuring efficient and secure blood donation, recipient management, and inventory tracking operations.*

## ▪ **Abstract:**

The Blood Bank Management System represents a pivotal advancement in healthcare infrastructure, offering a comprehensive solution for the efficient management of blood donation processes, recipient records, blood rates, and financial transactions within blood bank facilities. Developed using Java and MySQL, this system embodies the amalgamation of robust Object-Oriented Programming (OOP) principles and sophisticated database management techniques. By automating critical tasks and streamlining data management, the system addresses the pressing need for accuracy, accessibility, and timeliness in blood-related operations. With a user-friendly interface and a strong foundation in OOP design, the Blood Bank Management System stands as a testament to technological innovation in healthcare administration, poised to significantly enhance the efficacy and reliability of blood management processes in healthcare institutions.

## ▪ **History of Blood Bank Management Systems:**

The history of blood bank management systems traces back to the early 20th century, marked by the pioneering work of Dr. Bernard Fantus who established the first hospital blood bank in Chicago in 1937. This milestone ushered in a new era of medical practice, enabling the storage and transfusion of blood to save countless lives during emergencies and surgeries.

Over the decades, advancements in medical science and technology have revolutionized blood banking, transitioning from manual record-keeping to computerized systems. The advent of digital databases in the late 20th century significantly enhanced the efficiency and accuracy of blood management, allowing for seamless tracking of donor information, blood types, and inventory levels.

The 21st century witnessed a rapid evolution in blood bank management systems with the integration of sophisticated software solutions and database management techniques. These modern systems not only streamline the process of blood collection, testing, and storage but also facilitate real-time monitoring of blood inventory and donor records.

Furthermore, the emergence of mobile applications and online platforms has democratized blood donation, empowering individuals to register as donors, schedule appointments, and receive notifications about blood drives and shortages in their communities.

Today, blood bank management systems play a pivotal role in healthcare infrastructure worldwide, serving as the backbone of blood donation organizations, hospitals, and emergency response agencies. They ensure the seamless flow of blood from donors to recipients, optimizing resources, and ultimately saving lives in times of need.

## ▪ Objectives:

The primary objective of the Blood Bank Management System is to provide a robust platform for managing blood donation processes, recipient records, blood rates, and financial transactions. The system aims to automate and optimize various tasks associated with blood management to ensure timely availability of blood units for patients in need. By leveraging advanced technologies such as Java programming language and MySQL database management system, the system facilitates seamless coordination between blood donors, recipients, and healthcare providers. Additionally, the system incorporates features to enhance transparency and accountability in blood donation processes, including real-time tracking of donor availability, comprehensive recipient records management, and accurate monitoring of blood rates to reflect market dynamics. Moreover, the system prioritizes data security and privacy, implementing robust encryption and access control mechanisms to safeguard sensitive donor and recipient information. Through its intuitive user interface and customizable functionalities, the Blood Bank Management System empowers blood bank administrators and healthcare professionals to efficiently manage blood resources, optimize inventory levels, and respond promptly to patient needs, thereby contributing to the overall effectiveness and sustainability of blood donation and transfusion services.

## ▪ Scope:

### • Donor Management

Management of donor information, including personal details and blood donation history, to maintain a comprehensive database of potential blood donors.

### • Recipient Management

Recording and tracking recipient details and blood units received to ensure accurate matching of blood units with recipient needs.

### • Blood Rate Maintenance

Maintenance of blood rates for different blood groups to facilitate transparent pricing and effective management of blood resources.

- **Financial Transaction Handling**

Handling financial transactions related to blood donation and recipient services, ensuring secure and transparent payment processing.

- **Database Connectivity:**

Establish a connection with MySQL Workbench for data storage and retrieval.

- **User Authentication:**

Implement login and signup functionalities for authorized access.

- **GUI:**

Design interactive Graphical User Interfaces (GUI) using Java Swing components.

- **Beyond Current Focus:**

- **Blood Testing and Compatibility Checking**

Comprehensive blood testing and compatibility checking are not within the scope of the Blood Bank Management System and are handled separately by healthcare professionals.

- **Inventory Management of Non-Blood Supplies**

Management of medical supplies other than blood, such as medications and equipment, is beyond the scope of the system.

- **Integration with External Healthcare Systems**

Integration with external healthcare systems for broader healthcare management is not included in the scope of the Blood Bank Management System.

- **Blood Bank Management System –  
Functionality & Implementation**

- **Donor Management:**

- **Functionality:**

- Add New Donors:**

- Allow users to add new donors by entering their details such as first name, last name, phone number, gender, and hemoglobin level.

- Update Donor Information:**

- Provide functionality to update donor details such as phone number and hemoglobin level.

- View Donor Details:**

- Display a list of all donors with their respective details.

- **Implementation:**

- Java Classes:**

- AddDonor.java:** Implements functionality to add new donors.

- UpdateDonor.java:** Implements functionality to update donor information.

- ViewDetails.java:** Displays donor details.

- **Recipient Management:**

- **Functionality:**

- Add New Recipients:**

- Allow users to add new recipients by entering their details such as name and phone number.

- View Recipient Details:**

- Display a list of all recipients with their respective details.

- Request Blood:**

- Enable recipients to request blood units from the blood bank.

- **Implementation:**

- Java Classes:**

- AddRecipient.java:** Implements functionality to add new recipients.

- ViewRecipient.java:** Displays recipient details.

- Mainp.java:** Provides functionality for recipients to request blood units.

- **Blood Rate Management:**

- **Functionality:**

Set Blood Rates:

Allow authorized users to set and update blood rates based on blood groups.

- **Implementation:**

Java Class:

`BloodRate.java`: Implements functionality to set and update blood rates.

- **Billing:**

- **Functionality:**

**Generate Bills:**

Generate bills for blood units provided to recipients.

- **Implementation:**

Java Class:

***Payment.java***: Implements functionality to generate bills.

- **Database Connectivity:**

- **Functionality:**

**Establish Connection:**

Establish connection with MySQL Workbench for data storage and retrieval.

- **Implementation:**

Java Class:

***Conn.java***: Implements database connectivity using **JDBC** (Java Database Connectivity).



- **User Authentication:**

- **Functionality:**

- Login:**

- Authenticate users by validating their credentials.

- Signup:**

- Allow new users to register by providing their username and password.

- **Implementation:**

- Java Classes:**

- Login.java:** Implements user login functionality.

- SignUp.java:** Implements user signup functionality.

- **Graphical User Interface (GUI):**

- **Functionality:**

- Design GUI:**

- Design interactive GUI using Java Swing components for user interaction.

- **Implementation:**

- Java Classes:**

- Various classes implement GUI components for different functionalities.



# Entity Relationship Diagram

## Implementation:

In the Blood Bank Management System developed using Java in NetBeans and MySQL Workbench, the database schema is designed based on the Entity Relationship Diagram (ERD) to ensure efficient data organization and integrity. The ERD provides a visual representation of the database structure, including tables, attributes, and relationships between entities.

## Table Creation:

The ERD guides the creation of tables in the MySQL Workbench to represent various entities involved in the system. Each table corresponds to an entity identified in the ERD, and attributes are defined based on the ERD's entity-attribute relationships.

## Foreign Keys and Constraints:

Relationships between entities are established using foreign keys and constraints to maintain data integrity. Foreign keys are used to link tables together, representing associations between entities. Constraints such as primary keys, unique keys, and foreign key constraints ensure data consistency and enforce referential integrity within the database.

## For instance;

### ➤ Donorrr Table:

Attributes: Id (Primary Key), FirstName, LastName, PhoneNumber, Gender, HBLevel.

Primary Key: Id.

### ➤ Recipient Table:

Attributes: Id (Primary Key), Name, Phone, DonorId (Foreign Key), Units, BloodGroup.

Primary Key: Id.

Foreign Key: DonorId references Donorrr(Id).

➤ **BloodRate Table:**

Attributes: BloodGroup (Primary Key), RatePerUnit.

Primary Key: BloodGroup.

➤ **Bank Table:**

Attributes: Units, BloodGroup (Foreign Key).

Primary Key: (BloodGroup, Units).

Foreign Key: BloodGroup references BloodRate(BloodGroup).

➤ **Bill Table:**

Attributes: BillNo (Primary Key), Rid (Foreign Key), Amount.

Primary Key: BillNo.

Foreign Key: Rid references Recipient(Id).

➤ **Signup Table:**

Attributes: Username (Primary Key), Name, Password.

Primary Key: Username.

## **Explanation:**

The ERD serves as a blueprint for the database schema, ensuring that tables are properly designed to represent entities and their relationships accurately. By following the ERD during table creation and establishing relationships using foreign keys and constraints, the database maintains data consistency and integrity, crucial for the Blood Bank Management System's functionality.

## **Summary:**

The Entity Relationship Diagram (ERD) plays a vital role in the design and implementation of the database schema for the Blood Bank Management System. It provides a visual representation of entities and their relationships, guiding the creation of tables and ensuring data integrity through foreign keys and constraints. The ERD facilitates efficient data organization and retrieval, contributing to the system's overall effectiveness in managing blood bank operations.

# MySQL Database Queries

## Explanation:

The system interacts with a MySQL database using various queries to perform operations such as data retrieval, insertion, updating, and deletion. Notable queries include:

- Creation of tables and views
- Insertion procedures for adding recipients
- Trigger to check available units before recipient insertion
- Calculation of billing amount procedure

- **Implementation:**

In the Blood Bank Management System, MySQL Workbench is used to manage the database. Various queries are executed to create tables, views, stored procedures, triggers, and relationships. Stored procedures and triggers are utilized for data validation and manipulation.

```
1 • create database bloodbanksystem;
2 • use bloodbanksystem;
3
4 • select * from donorr;
5 • select * from recipient;
6 • select * from bloodrate;
7 • select * from bank;
8 • select * from bill;
9 • select * from signup;
10
```

- **Create Database:**

CREATE DATABASE bloodbanksystem;

- **Use Database:**

USE bloodbanksystem;

- **Create Tables:**

## Explanation:

These queries are used to create tables in the database for storing donor, recipient, blood rate, bank, bill, and signup information.

```
CREATE TABLE donorr (
    Id INT PRIMARY KEY,
    FirstName VARCHAR(255),
    LastName VARCHAR(255),
    PhoneNumber VARCHAR(15),
    Gender VARCHAR(10),
    HbLevel VARCHAR(255)
);
```

Result Grid								
Filter Rows:								
Edit:								
Export/Import:								
	id	firstname	lastname	phonenumber	bloodgroup	gender	units	hblevel
▶	1	aslam	sipra	0332788975	B+	Male	3	15
	2	rukhsana	alam	04245526776	AB+	Female	4	13
	3	Khansa	Jan	0324929834	B-	Male	2	13
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

```
CREATE TABLE recipient (
    Id INT PRIMARY KEY,
    Name VARCHAR(255),
    Phone VARCHAR(15),
    Units INT,
    BloodGroup VARCHAR(3),
    DonorId INT,
    FOREIGN KEY (DonorId) REFERENCES donorr(Id)
);
```

	id	name	phone	donorid	units	bloodgroup
▶	1	Shahana	03387483847	14	5	NULL
	2	Tariq	0493943949	13	4	A-
	3	Khudaim	02337283728	16	2	O+
	4	shahan	02327328738	16	2	O+

```
CREATE TABLE bloodRate (
    BloodGroup VARCHAR(3) PRIMARY KEY,
    RatePerUnit FLOAT
);
```

	bloodgroup	rateperunit
▶	A+	40
	A-	40
	AB+	20
	AB-	3
	B+	3
	B-	45
	O+	4
	O-	50
*	NULL	NULL

```

CREATE TABLE bank (
    Units INT,
    BloodGroup VARCHAR(3),
    PRIMARY KEY (BloodGroup, Units),
    FOREIGN KEY (BloodGroup) REFERENCES bloodRate(BloodGroup)
);

```

	bloodgroup	units
▶	A+	0
	A-	0
	AB+	4
	AB-	0
	B+	3
	B-	2
	O+	0
	O-	0
*	NULL	NULL

```

CREATE TABLE bill (
    BillNo INT AUTO_INCREMENT PRIMARY KEY,
    Rid INT,
    Amount INT,
    FOREIGN KEY (Rid) REFERENCES recipient(Id)
);

```

	billno	Rid	amount
▶	1	2	172
	2	3	8
	3	3	8
	4	8	250
	5	11	80
	6	2	160
	7	35	8
	8	36	160
	PRIMARY	PRIMARY	PRIMARY

```

CREATE TABLE signup (
    Username VARCHAR(255) PRIMARY KEY,
    Name VARCHAR(255),
    Password VARCHAR(255)
);

```

	name	username	password	occupation
	newbtn	newbtnktk	newbtn123	Nurse
	qureshi	qureshiktk	qureshi123	Patient
	Final	finaltk	final123	Patient
	shayan	shayan123	shayanktk	Nurse
	shan	shanktk	shan123	Nurse
	shandana	shandana123	shandana	Nurse
	s	s	s	Admin
	shak	shak@bbs.c...	shak	Admin
	hafsa	haf@bbs.com	haf	Admin

- **Create View:**

### Explanation:

This query creates a view named DisplayTable to display recipient details along with donor information.

CREATE VIEW DisplayTable AS

SELECT r.Name AS 'Recipient Name', r.Id AS 'Recipient ID', d.Id AS 'Donor ID', d.FirstName AS 'Donated by',  
r.Units AS 'Units Purchased', r.BloodGroup AS 'Blood Group'

FROM donorr AS d

JOIN recipient AS r ON d.Id = r.DonorId;

```
create view DisplayTable as
select r.name as 'Recipient Name', r.id as 'Recipient ID', d.id as 'Donor ID', d.firstname as 'Donated by', r.units as 'Units purchased',
r.bloodgroup as 'BloodGroup' from donorr as d join recipient as r on d.id = r.donorid;
select * from DisplayTable;
alter table donorr add constraint fk2fs foreign key(bloodgroup, units) references bank(bloodgroup, units);
alter table bank add constraint pk1fs primary key(bloodgroup, units);
```

- **Stored Procedure CREATE PROCEDURE InsertRecipient**

### Explanation:

Stored procedures are used for encapsulating business logic that can be reused within the database.

DELIMITER //

CREATE PROCEDURE InsertRecipient(

IN p\_name VARCHAR(255),

IN p\_phone VARCHAR(15),

IN p\_donorid INT,

IN p\_units INT,

IN p\_bloodgroup VARCHAR(3)

)

BEGIN

DECLARE donor\_units INT;

SELECT Units INTO donor\_units

FROM donorr

WHERE Id = p\_donorid;



```

IF p_units > donor_units THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Requested units exceed donor's available units.';
ELSE
    INSERT INTO recipient(Name, Phone, DonorId, Units, BloodGroup)
    VALUES (p_name, p_phone, p_donorid, p_units, p_bloodgroup);

    UPDATE donorr
    SET Units = Units - p_units
    WHERE Id = p_donorid;
    SELECT 'Recipient Added Successfully' AS Result;
END IF;
END //

DELIMITER ;

```

```

CREATE PROCEDURE InsertRecipient(
    IN p_name VARCHAR(255),
    IN p_phone VARCHAR(15),
    IN p_donorid INT,
    IN p_units INT,
    IN p_bloodgroup VARCHAR(3)
)
BEGIN
    DECLARE donor_units INT;

    -- Check if requested units are greater than donor units
    SELECT units INTO donor_units
    FROM donorr
    WHERE id = p_donorid;

    IF p_units > donor_units THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Requested units exceed donor's available units.';
    ELSE
        -- Continue with saving recipient information
        INSERT INTO recipient(name, phone, donorid, units, bloodgroup)
        VALUES (p_name, p_phone, p_donorid, p_units, p_bloodgroup);
    END IF;
END

```

```

-- Update donor units
UPDATE donorrrr
SET units = units - p_units
WHERE id = p_donorid;

-- Display a message or handle further logic
SELECT 'Recipient Added Successfully' AS Result;
END IF;
END //

```

- CREATE TRIGGER check\_units\_before\_insert

### Explanation:

Triggers are used to automatically perform actions in response to certain events on tables.

```

DELIMITER //

CREATE TRIGGER check_units_before_insert
BEFORE INSERT ON recipient
FOR EACH ROW
BEGIN
    DECLARE donor_units INT;

    SELECT Units INTO donor_units FROM donorrrr WHERE Id = NEW.DonorId;

    IF NEW.Units > donor_units THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Insufficient amount of units for selected donor.';
    END IF;
END;

DELIMITER ;

```

```

CREATE TRIGGER check_units_before_insert
BEFORE INSERT ON recipient
FOR EACH ROW
BEGIN
    DECLARE donor_units INT;

    -- Retrieve donor units for the specified donor ID
    SELECT units INTO donor_units FROM donorrrr WHERE id = NEW.donorid;

    -- Check if requested units are greater than donor units
    IF NEW.units > donor_units THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Insufficient amount of units for selected donor.';
    END IF;
END;

DELIMITER;

DELIMITER //

```

- Stored Procedure CREATE PROCEDURE CalculateAmount

### Explanation:

This stored procedure calculates the amount based on units and rate per unit.

```
DELIMITER //
```

```
CREATE PROCEDURE CalculateAmount(IN p_units INT, IN p_rate INT, OUT p_amount INT)
```

```
BEGIN
```

```
    SET p_amount = p_units * p_rate;
```

```
END//
```

```
DELIMITER ;
```

```

CREATE PROCEDURE CalculateAmount(IN p_units INT, IN p_rate INT, OUT p_amount INT)
BEGIN
    SET p_amount = p_units * p_rate;
END//

-- DELIMITER ;
select * from donorrrr;

select * from recipient;

```



**JAVA**

# Java Classes



- *BloodBankSystem.java*

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Main.java to edit this template
 */
package bloodbanksystem;

public class BloodBankSystem {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

}
```

- *Conn.java*

```
package bloodbanksystem;

import java.sql.*;

public class Conn
{
    Connection c;
    Statement s;

    Conn() {
        try{
            Class.forName("com.mysql.cj.jdbc.Driver");
            c = DriverManager.getConnection("jdbc:mysql:///bloodbanksystem", "root", "");
            s = c.createStatement();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

## • adddonor.java

```
package bloodbanksystem;

import java.sql.PreparedStatement;
import javax.swing.JOptionPane;
import javax.swing.*;
import java.awt.*;
import java.sql.*;
import java.awt.event.*;
import static java.lang.Integer.parseInt;

public class adddonor extends javax.swing.JFrame {

    public adddonor() {
        initComponents();
    }

    @SuppressWarnings("unchecked")
    // Generated Code

    private void backbtnActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        setVisible(false);
        mainp su = new mainp();
        su.setVisible(true);
    }

    private void savebtnActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        String firstName = tffirstname.getText();
        String lastName = tflastname.getText();
        String phoneNumber = tfphonenumber.getText();
        String bloodGroup = (String) tfbloodgroup.getSelectedItem();
        String gender = (String) tfgender.getSelectedItem();
        String unitString = tfunit.getText();
        String hbString = tfhb.getText();

        // Check for null or empty values
        if (firstName.isEmpty() || lastName.isEmpty() || phoneNumber.isEmpty() || unitString.isEmpty() || hbString.isEmpty()) {
            JOptionPane.showMessageDialog(null, "Please fill in all the required fields.");
            return;
        }

        // Phone number validation using regular expression
        String phoneNumberRegex = "[0-9]+$";
        if (!phoneNumber.matches(phoneNumberRegex)) {
            JOptionPane.showMessageDialog(null, "Invalid phone number. Please enter only numeric digits.");
            return; // Stop further execution if the phone number is invalid
        }

        int hblevel = parseInt(tfhb.getText());
        int unit = parseInt(tfunit.getText());
        String bloodgroup = (String) tfbloodgroup.getSelectedItem();

        if (hblevel >= 10)
        {
            try
            {
                Conn con = new Conn();

                String sql = "insert into donorr(firstname, lastname, phonenumber, bloodgroup, gender, units, hblevel) values(?, ?, ?, ?, ?, ?, ?)";
                PreparedStatement st = con.c.prepareStatement(sql);
```

## • *bloodrate.java*

```
try{
    Conn conn = new Conn();

    String sql= "select * from bloodRate";
    PreparedStatement st = conn.c.prepareStatement(sql);
    ResultSet rs= conn.s.executeQuery(sql);
    DefaultTableModel tm=(DefaultTableModel)studTable.getModel();
    tm.setRowCount(0);
    while(rs.next()){
        Object o[] = {rs.getString("bloodgroup"),rs.getFloat("rateperunit")};
        tm.addRow(o);
    }

} catch (Exception e){
    JOptionPane.showMessageDialog(null, e);
}

}

private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    Float rateperunit = parseFloat(tftrate.getText());
    String bloodgroup = tfbloodgroup.getText();

    try
    {
        Conn con = new Conn();

        String q1 = "update bloodRate set rateperunit = '"+rateperunit+"'where bloodgroup = '"+bloodgroup+"'";
        con.s.executeUpdate(q1);

        JOptionPane.showMessageDialog(null, "Rate Updated Successfully");
    }
}
```

## • *deletedonor.java*

```
try
{
    Conn con = new Conn();

    String donorid = tfdonorid.getText();
    String bloodgroup = null;
    int unit = 0;

    ResultSet rs = con.s.executeQuery("select * from donorrr where id = '"+donorid+"'");
    if(rs.next())
    {
        bloodgroup = rs.getString(5);
        unit = rs.getInt(7);
    }

    String q1 = "update bank set units = units - " + unit + " where bloodgroup = '" + bloodgroup + "'";
    con.s.executeUpdate(q1);

    JOptionPane.showMessageDialog(null, "Bank Updated Successfully");

    String sql2 = "DELETE FROM donorrr WHERE id='"+donorid+"'";
    PreparedStatement st2= con.c.prepareStatement(sql2);
    st2.executeUpdate();
    JOptionPane.showMessageDialog( null, "Donor has been removed");
} catch (Exception e){
    System.out.println(e);
}
}
```

## • *login.java*

```
try {
    Conn con = new Conn();
    String sql = "select * from signup where username=? and password=?";
    PreparedStatement st = con.c.prepareStatement(sql);

    st.setString(1, tfusername.getText());
    st.setString(2, tfpassword.getText());

    ResultSet rs = st.executeQuery();
    if (rs.next()) {
        setVisible(false);
        mainp mu = new mainp();
        mu.setVisible(true);
    } else
        JOptionPane.showMessageDialog(null, "Invalid Login or Password!");
} catch (Exception e2) {
    e2.printStackTrace();
}
}
```

- *mainp.java*

```
private void addpatientbtnActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    setVisible(false);
    adddonor su = new adddonor();
    su.setVisible(true);
}

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    setVisible(false);
    viewdetails su = new viewdetails();
    su.setVisible(true);
}

private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    setVisible(false);
    reqforblood su = new reqforblood();
    su.setVisible(true);
}

private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    setVisible(false);
    updatedonor su = new updatedonor();
    su.setVisible(true);
}

private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    setVisible(false);
    viewrecipient su = new viewrecipient();
    su.setVisible(true);
}
}
```

- *payment.java*

```
try
{
    Conn con = new Conn();

    ResultSet rs = con.s.executeQuery("select * from recipient where id = '"+rid+"'");
    if(rs.next())
    {
        bloodgroup = rs.getString(6);
        units = rs.getInt(5);
    }

    ResultSet rs2 = con.s.executeQuery("select * from bloodRate where bloodgroup = '"+bloodgroup+"'");
    if(rs2.next())
    {
        rate = rs2.getInt(2);
    }

    int amount = units * rate; // Calculate the amount as an integer

    tfamount.setText(String.valueOf(amount));
    all the stored procedure to calculate the amount
    String calculateAmountProcedure = "{CALL CalculateAmount(?, ?, ?)}";
    try (CallableStatement cs = (CallableStatement) con.c.prepareCall(calculateAmountProcedure)) {
        cs.setInt(1, units);
        cs.setInt(2, rate);
        cs.registerOutParameter(3, Types.INTEGER);
        cs.execute();
        amount = cs.getInt(3);

        tfamount.setText(String.valueOf(amount));
    }
    catch (Exception e)
    {
        System.out.println(e);
    }
}
```

```

        catch(Exception e)
        {
            System.out.println(e);
        }

        String sql = "insert into bill(Rid, amount) values(?, ?)";

        PreparedStatement st = con.c.prepareStatement(sql);

        st.setInt(1, rid); // Use setInt for integer values
        st.setInt(2, amount);

        ResultSet rs3 = con.s.executeQuery("select * from recipient where id = '"+rid+"'");
        if(rs3.next()){
            tfbloodgroup.setText(rs3.getString(6));
            tfunits.setText(rs3.getString(5));
            tfname.setText(rs3.getString(2));
        }

        int i = st.executeUpdate();
        if (i > 0) {
            JOptionPane.showMessageDialog(null, "Bill Made Successfully.");
        }

        ResultSet rs9;
        rs9 = con.s.executeQuery("select * from bill where amount = '"+amount+"'");
        if(rs9.next()){
            tfbill.setText(String.valueOf(rs9.getInt(1)));
        }

    } catch(Exception e){
        System.out.println(e);
    }

```

## • reqforblood.java

```

try {
    int donorId = Integer.parseInt(donorIdText);
    int units = Integer.parseInt(unitText);

    // Check if the donor ID exists in the donorrr table
    Conn con = new Conn();
    String donorQuery = "SELECT * FROM donorrr WHERE id = ?";
    PreparedStatement donorSt = con.c.prepareStatement(donorQuery);
    donorSt.setInt(1, donorId);
    ResultSet donorRs = donorSt.executeQuery();

    if (!donorRs.next()) {
        JOptionPane.showMessageDialog(null, "Donor ID does not exist in the donorrr table.");
        return; // Stop further execution if the donor ID does not exist
    }

    int dunits = donorRs.getInt("units");

    // Check if requested units are greater than donor units
    if (units > dunits) {
        JOptionPane.showMessageDialog(null, "Requested units exceed donor's available units.");
        return; // Stop further execution if requested units exceed donor's units
    }

    // Continue with saving recipient information
    String recipientQuery = "INSERT INTO recipient(name, phone, donorid, units, bloodgroup) VALUES (?, ?, ?, ?, ?)";
    PreparedStatement st = con.c.prepareStatement(recipientQuery);

```



## • *signup.java*

```
private void createActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    String name = tfname.getText();
    String username = tfusername.getText();
    String password = tfpassword.getText();
    String occupation = (String) tfoccupation.getSelectedItem();

    String usernameRegex = "^[a-zA-Z0-9]+@bbs\\.com$";
    if (!username.matches(usernameRegex)) {
        JOptionPane.showMessageDialog(null, "Invalid username format. It must be in the format xxx@bbs.com");
        return; // Stop further execution if the username is invalid
    }

    String query = "insert into signup values('" + name + "', '" + username + "', '" + password + "', '" + occupation + "')";

    try {
        Conn c = new Conn();
        c.s.executeUpdate(query);
        JOptionPane.showMessageDialog(null, "Account Created Successfully");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

## • *splash.java*

```
public class splash {
    public static void main(String[] args) {
        SplashFrame f1 = new SplashFrame();
        f1.setVisible(true);
        int i;
        int x=1;
        for(i=2; i<=600; i+=10, x+=7){
            f1.setLocation(700 - ((i+x)/2), 380 - (i/2));
            f1.setSize(i+x,i);
            try {
                Thread.sleep(10);
            } catch (Exception e) {}
        }
    }
}

class SplashFrame extends JFrame implements Runnable {
    Thread t1;
    SplashFrame() {
        setLayout(new FlowLayout());
        ImageIcon c1 = new ImageIcon(ClassLoader.getResource("bloodbanksystem/icons/Splash1.png"));
        Image i1 = c1.getImage().getScaledInstance(1050, 680, Image.SCALE_DEFAULT);
        ImageIcon i2 = new ImageIcon(i1);
    }
}
```

## • *stock.java*

```
private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    Conn c = new Conn();

    try {
        String q1 = "update bank set units = '"+0+"' where bloodgroup in ('A+', 'A-', 'B+', 'B-', 'O+', 'O-', 'AB+', 'AB-') ";
        c.s.executeUpdate(q1);

        JOptionPane.showMessageDialog(null, "Stock cleared Successfully");
    } catch (Exception e1) {
        System.out.println(e1.getMessage());
    }
}
```

- *updatedonor.java*

```
try{
    String s1 = tffirstname.getText();
    String s2 = tflastname.getText();
    String s3 = tfphonenumber.getText();
    String s4 = tfbloodgroup.getText();
    String s5 = tfgender.getText();

    String q1 = "update donorr set firstname = '"+s1+"', lastname = '"+s2+"', phonenumber = '"+s3+"', bloodgroup = '"+s4+"', gender = '"+s5+"' where id = '"+id+"'";
    c.s.executeUpdate(q1);

    JOptionPane.showMessageDialog(null, "Customer Detail Updated Successfully");
}catch(Exception e1){
    System.out.println(e1.getMessage());
}
```

- *viewdetails.java*

```
try{
    boolean donorExists = checkDonorExistence(id);

    if (!donorExists) {
        JOptionPane.showMessageDialog(null, "Donor ID does not exist.");
    }

    else
    {
        Conn c = new Conn();
        ResultSet rs = c.s.executeQuery("select * from donorr where id = '"+id+"'");
        if(rs.next()){
            tffirstname.setText(rs.getString(2));
            tflastname.setText(rs.getString(3));
            tfphonenumber.setText(rs.getString(4));
            tfbloodgroup.setText(rs.getString(5));
            tfgender.setText(rs.getString(6));
            tfunit.setText(rs.getString(7));
        }
    }

}catch(Exception e){ }
```

- *viewreceptient.java*

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {

    try{
        Conn conn = new Conn();

        String sql= "select r.name as 'Recipient Name', r.id as 'Recipient ID', r.donorid as 'Donor ID',"
            + " d.firstname as 'Donoted by', r.units as 'Units purchased', r.bloodgroup as 'BloodGroup'"
            + " from donorr as d join recipient as r on d.id = r.donorid";
        PreparedStatement st = conn.c.prepareStatement(sql);
        ResultSet rs= conn.s.executeQuery(sql);
        DefaultTableModel tm=(DefaultTableModel)studTable.getModel();
        tm.setRowCount(0);
        while(rs.next()){
            Object o[] = {rs.getString("Recipient Name"),rs.getInt("Recipient ID"),rs.getInt("Donor ID"),
                rs.getString("Donoted by"),rs.getInt("Units purchased"),
                rs.getString("BloodGroup")};
            tm.addRow(o);
        }

    }catch (Exception e){
        JOptionPane.showMessageDialog(null, e);
    }
}
```

## • About.java

```
package bloodbanksystem;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.Font;
import javax.swing.JFrame;
import javax.swing.border.Border;
import java.awt.Color;

public class About extends JFrame implements ActionListener {

    JButton b1;
    JLabel l1;
    Font f, f1, f2;
    TextArea t1;
    String s;

    public About() {

        setLayout(null);
        JButton b1 = new JButton("Back");
        add(b1);
        b1.setBounds(260, 430, 120, 20);
        b1.addActionListener(this);

        Font f = new Font("RALEWAY", Font.BOLD, 180);
        setFont(f);

        s = " Welcome to our Blood Bank System,\n "
            + "\na compassionate platform designed to bridge the gap between donors and recipients in "
            + "times of critical need. Our system is dedicated to saving lives by "
            + "facilitating the seamless exchange of life-saving blood units. With "
            + "the ability to add new donors, request specific blood types, and "
            + "view a comprehensive list of generous donors. \n\n"
            + "Our meticulously curated blood rate list ensures transparency and "
            + "fairness, while bills are generated effortlessly when recipients purchase blood. \n\n"

    }

    public void actionPerformed(ActionEvent e) {
        setVisible(false);
        mainp su = new mainp();
        su.setVisible(true);
    }

    public static void main(String args[]) {
        new About().setVisible(true);
    }
}
```

## Explanation:

### • BloodBankSystem.java:

**Functionality:** Initializes the Blood Bank Management System.

**Implementation:**

Contains the main method to start the application.

May include initialization of essential components or resources.

- **Conn.java:**

**Functionality:** Handles database connectivity.

**Implementation:**

Establishes a connection to the MySQL Workbench database using JDBC.

Provides methods for connecting to and disconnecting from the database.

May include error handling for database connection failures.

- **adddonor.java:**

**Functionality:** Adds new donors to the system.

**Implementation:**

Utilizes Java Swing components to create a user-friendly form for entering donor information such as name, phone number, gender, etc.

Validates user input before storing the data in the database.

May include error handling for database insertions.

- **bloodrate.java:**

**Functionality:** Manages viewing and updating of blood rates.

**Implementation:**

Retrieves blood rate information from the database and displays it using Java Swing GUI components.

Provides functionality to update blood rates, including error handling for invalid inputs.

May include logging or auditing mechanisms for rate updates.

- **deletedonor.java:**

**Functionality:** Deletes donor records from the system.

**Implementation:**

Retrieves a list of donors from the database and provides a user interface for selecting and deleting donor records.

Implements confirmation dialogs to prevent accidental deletions.

Handles database deletion operations and updates UI accordingly.

- **login.java:**

**Functionality:** Handles user authentication for system login.

**Implementation:**

Presents a login form for users to enter their credentials.

Validates user inputs against records stored in the database.

Grants access to authorized users and displays appropriate error messages for invalid credentials.

- **mainp.java:**

**Functionality:** Serves as the main dashboard of the application.

**Implementation:**

Displays a user-friendly interface with navigation options for accessing various modules of the Blood Bank Management System.

May include dynamic content based on user roles or permissions.

Implements event handling for user interactions with the dashboard components.

- **payment.java:**

**Functionality:** Manages payment processing for requested blood units.

**Implementation:**

Calculates the total amount to be paid based on the requested blood units and rate per unit.

Generates bills or invoices for payment and stores them in the database.

May include integration with payment gateways for online transactions.

- **reqforblood.java:**

**Functionality:** Facilitates recipients in requesting blood units from the blood bank.

**Implementation:**

Presents a form for recipients to specify their blood requirements.

Sends requests to the blood bank module for processing and fulfillment.

May include real-time status updates on request processing.

- **signup.java:**

**Functionality:** Implements user registration/signup functionality.

**Implementation:**

Provides a registration form for new users to create an account.

Validates user inputs and checks for duplicate usernames.

Stores user registration details securely in the database.

- **splash.java:**

**Functionality:** Displays a splash window during application startup.

**Implementation:**

Shows a visually appealing splash screen with relevant branding or loading messages.

May include timers to control the duration of the splash screen.

Provides a smooth transition to the main application window upon completion.

- **stock.java:**

**Functionality:** Manages viewing and checking stocks of blood units in the blood bank.

**Implementation:**

Retrieves real-time data on available blood stocks from the database.

Displays the current stock levels using Java Swing components.

May include visual indicators for low stock alerts.

- **updatedonor.java:**

**Functionality:** Implements functionality to update donor details in the system.

**Implementation:**

Retrieves donor records from the database for updating.

Provides a user interface to modify donor information such as phone number or hemoglobin level.

Handles database update operations and validates user inputs.

- **viewdetails.java:**

**Functionality:** Displays comprehensive details of donors.

**Implementation:**

Retrieves detailed donor information from the database, including personal and medical details.

Presents the information in a user-friendly format using Java Swing GUI components.

May include search and filtering options for navigating through donor records.

- **viewrecepient.java:**

**Functionality:** Displays comprehensive details of recipients.

**Implementation:**

Retrieves detailed recipient information from the database, including contact details and blood requirements.

Presents the information in a clear and organized manner using Java Swing GUI components.

May include features for sorting or categorizing recipient records.

- **About.java:**

**Functionality:** Provides information about the Blood Bank Management System.

**Implementation:** Displays details such as the purpose of the system, its features, and any relevant acknowledgments or credits.

Utilizes Java Swing components to create a visually appealing interface for presenting information.

May include dynamic content updates to reflect changes or updates to the system.

Implements event handling for user interactions with the About page, such as closing the window or returning to the main menu.

## ▪ Input and Output Specifications

### ➤ Input Specifications:

- **Donor Information Input:**

First name, last name, phone number, gender, and hemoglobin level when adding a new donor.

Updates to donor information such as phone number or hemoglobin level.

- **Recipient Information Input:**

Name and phone number when adding a new recipient.

Blood group and number of units required when requesting blood.

- **Blood Rate Input:**

Blood group and rate per unit when setting or updating blood rates.

- **Payment Input:**

Number of blood units requested.

Payment method (e.g., cash, card) during payment processing.

- **Login Information Input:**

Username and password during user authentication.

- **Signup Information Input:**

Username, name, and password during user registration.

- **Database Query Input:**

SQL queries for database operations such as selecting, updating, inserting, or deleting records.

### ➤ Output Specifications:

- **Donor Information Output:**

List of donors with their details (first name, last name, phone number, gender, hemoglobin level).

Updated donor details confirmation messages.

- **Recipient Information Output:**

List of recipients with their details (name, phone number).

Confirmation messages for adding recipients or requesting blood.

- **Blood Rate Output:**

Blood group and corresponding rate per unit.

Confirmation messages for updating blood rates.

- **Billing Output:**

Generated bills or invoices for payment.

Confirmation messages for successful payments.

- **Login Output:**

Successful login confirmation message.

Error messages for invalid credentials.

- **Signup Output:**

Confirmation message for successful registration.

Error messages for duplicate usernames or invalid inputs.

- **Database Query Output:**

Results of SQL queries executed (e.g., records retrieved, updated, inserted, or deleted).

Error messages for failed database operations.

- **GUI Output:**

Graphical User Interface displaying various modules and functionalities.

User-friendly interfaces for data input and output across different screens.



## ▪ Usage of OOP Concepts in Blood Bank Management System

### **Classes and Objects:**

The Blood Bank Management System employs various classes such as Donor, Recipient, BloodRate, DatabaseConnector, and BloodBankManagementSystem.

Objects of these classes represent entities within the blood bank system, such as donors, recipients, blood rates, and database connections.

### **Inheritance:**

Inheritance establishes relationships between classes in the Blood Bank Management System.

For example, a base class like Person can be created, extended by subclasses like Donor and Recipient to inherit common attributes and behaviors related to personal information.

### **Encapsulation:**

Encapsulation is applied to classes in the Blood Bank Management System to encapsulate data and functionality.

For instance, the Donor class encapsulates donor-specific details like first name, last name, phone number, and gender, providing methods to securely access and modify this information.

### **Method Overriding:**

Method overriding is implemented when custom implementations of inherited methods are needed in the Blood Bank Management System.

For example, the displayDetails() method in the Donor class can override the same method in a superclass to display donor-specific information.

### **Constructor:**

Constructors are vital components in various classes of the Blood Bank Management System to initialize object properties during their creation.

For example, a constructor in the DatabaseConnector class can be utilized to set initial values such as connection details, ensuring proper database connectivity.

### **Polymorphism:**

Polymorphism is applied in the Blood Bank Management System to treat objects of different classes as instances of a common superclass or interface.

For flexibility and code reusability, methods that process blood transactions can accept both Donor and Recipient objects, treating them as instances of a common BloodItem superclass.

**By leveraging these Object-Oriented Programming (OOP) principles, the Blood Bank Management System ensures a robust and maintainable codebase. This approach facilitates efficient management of blood resources and interactions within the system, promoting flexibility and scalability.**

## ▪ **Software/Equipment Used:**

- **Development Environment:**

NetBeans IDE: Used for Java development, providing a comprehensive environment for coding, debugging, and building Java applications.

- **Database Management System:**

MySQL Workbench: Utilized as the database management system for storing and managing data related to the Blood Bank Management System.

- **Programming Languages and Technologies:**

Java: Primary programming language used for developing the Blood Bank Management System, leveraging its Object-Oriented Programming (OOP) features and libraries.

JDBC (Java Database Connectivity): Used for database connectivity in Java applications, enabling interaction with the MySQL database in the Blood Bank Management System.

- **Graphics and User Interface:**

Java Swing: Employed for designing and developing the graphical user interface (GUI) components of the Blood Bank Management System, providing a platform-independent approach for creating interactive interfaces.

- **Documentation:**

Microsoft Word: Utilized for creating detailed documentation, including project specifications, user manuals, and technical documentation.

- **Version Control:**

Git: Employed for version control, allowing collaborative development, tracking changes, and managing the project's source code repository.

- **Operating System:**

Windows/Linux/macOS: The Blood Bank Management System is designed to be compatible with multiple operating systems, ensuring accessibility and flexibility for users.

- **Hardware Requirements:**

Personal Computer: A standard desktop or laptop computer with sufficient processing power and memory to support Java development and MySQL database operations.

- **Output and Execution Flow in Blood Bank Management System:**

- **Splash Screen:**

Upon launching the application, the splash screen (implemented in Splash.java) is displayed, indicating that the application is loading.

- **Login Page:**

After the splash screen, the user is directed to the login page (implemented in Login.java).

The user enters their username and password to authenticate.

- **Main Dashboard:**

Upon successful login, the user is redirected to the main dashboard (implemented in Mainp.java).

The main dashboard serves as the central hub for accessing various functionalities of the blood bank management system.

- **Functionality Modules:**

From the main dashboard, the user can navigate to different functionality modules represented by buttons or tabs.

Each functionality module corresponds to a specific Java class, such as AddDonor.java, UpdateDonor.java, ViewDetails.java, ViewRecipient.java, BloodRate.java, Stock.java, DeleteDonor.java, Payment.java, etc.

- **Input and Interaction:**

Within each functionality module, the user interacts with the graphical user interface (implemented using Java Swing components) to perform various tasks.

For example, in the Add Donor module, the user inputs donor details such as first name, last name, phone number, gender, etc., and clicks on the "Add Donor" button to add a new donor.

➤ **Database Interaction:**

The Java classes responsible for database connectivity (e.g., Conn.java) handle interactions with the MySQL Workbench database.

SQL queries are executed to retrieve, insert, update, or delete data from the database tables such as donorrrr, recipient, bloodrate, bank, bill, signup, etc.

➤ **Processing and Output:**

Based on user inputs and database interactions, the system processes the requested actions.

For example, when a new donor is added, the system validates the input data, inserts the donor's information into the database, and displays a confirmation message indicating the successful addition of the donor.

➤ **Error Handling:**

The system includes error handling mechanisms to deal with exceptions and unexpected behaviors gracefully.

For instance, if the user tries to add a donor with invalid input data, appropriate error messages are displayed to prompt the user to correct the input fields.

➤ **Navigation and Feedback:**

Throughout the execution flow, the system provides navigation options such as a "Back" button to allow users to return to the main dashboard or previous screens.

Feedback messages and notifications are displayed to inform users about the outcome of their actions, such as successful operations, errors, or warnings.

➤ **Logout and Termination:**

The system does not have explicit logout or exit options.

Users can navigate back to the main dashboard using the "Back" button or close the application window to terminate the program execution.

- Output Display:

❖ *splash.java*



❖ *signup.java*

The image is a screenshot of a web application window titled 'SIGN UP RN!'. The window has a red background. It contains four input fields: 'Name' with the value 'shaz', 'Username' with the value 'shaz@gmail.com', 'Password' with the value 'shaz', and 'Occupation' with a dropdown menu showing 'Admin'. Below the fields are two buttons: 'Create Account' and 'Back'. At the bottom of the window, there is a 'Message' dialog box with a blue information icon and the text 'Invalid username format. It must be in the format xxx@bbs.com'. The dialog box has an 'OK' button.

❖ *login.java*



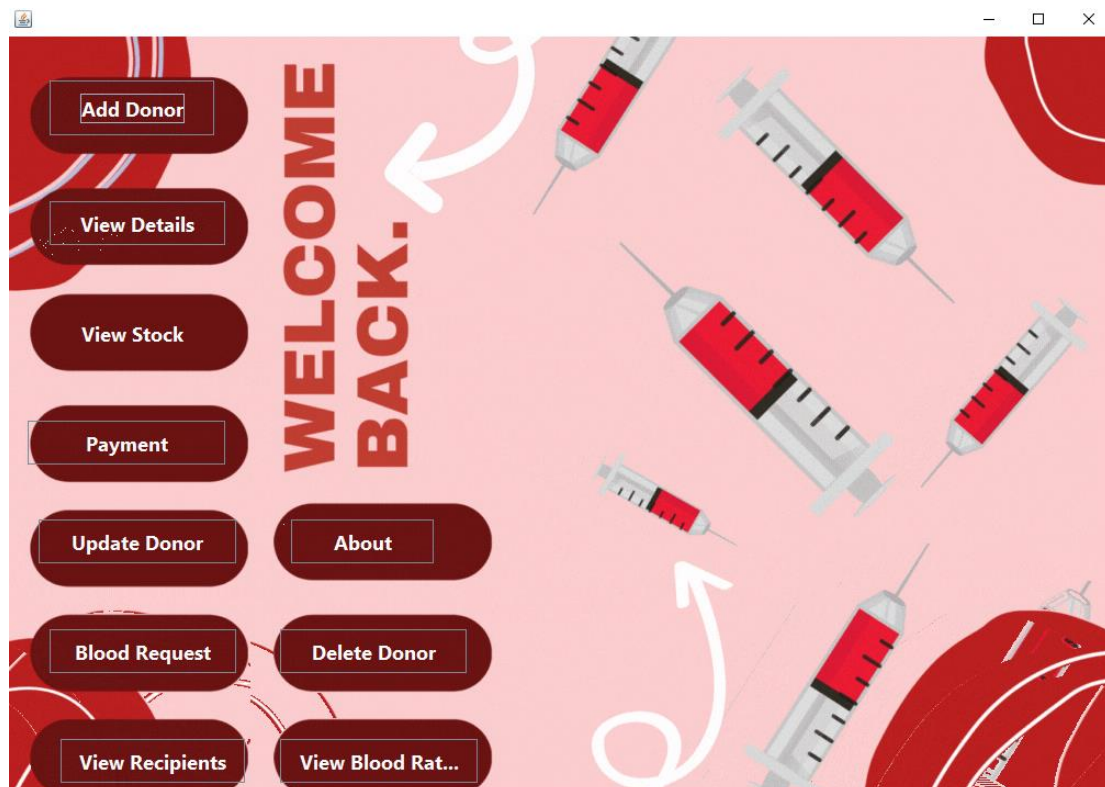
**BLOOD BANK**

Username:

Password:

Login Sign Up

❖ *mainp.java*



**WELCOME BACK.**

Add Donor

View Details

View Stock

Payment

Update Donor

About

Blood Request

Delete Donor

View Recipients

View Blood Rat...

## Add New Donor

First Name:

Last Name:

Phone Number:

Blood Group:

Units:  \*1 unit = 5ml

HB:

Gender:

Your ID:

Message

You cannot donate blood

## Add New Donor

First Name:

Last Name:

Phone Number:

Blood Group:

Units:  \*1 unit = 5ml

HB:

Gender:

Your ID:



## ❖ *viewdetails.java*

**Details of Donors**

Enter your ID:

Units Donated:

First Name:

Last Name:

Phone Number:

Blood Group:

Gender:

## ❖ *stock.java*

**Available Units**

Display Stock	
Blood Type	Units
A+	10
A-	0
AB+	5
AB-	0
B+	0
B-	5
O+	0
O-	0



## ❖ *updatedonor.java*

**Update Donor**

Enter Donor ID:  [View Details](#)

First Name:

Last Name:

Phone Number:

Blood Group:

Gender:

[Save](#) [Back](#)

**Message**

Customer Detail Updated Successfully

[OK](#)

## ❖ *deletedonor.java*

**Delete Donor**

Enter Donor ID:  [Delete](#)

[Back](#)

**Message**

Bank Updated Successfully

[OK](#)

## ❖ reqforblood.java

Request for Blood

Enter Required Blood Group: B-

View Donors

ID	FName	LName	Phone number	Blood Group	Gender	Units
5	Sharjeel	Cp	034023490	B-	Male	5

Message

Recipient Added Successfully.

OK

Choose Donor ID: 5

Name: Ladain

Phone: 032934289

Units Required: 5

Your Recipient ID is:

Save Back

## ❖ viewreceptient.java

View Recipient

Recipient Na...	Recipient ID	Donor ID	Donated by	Units Purcha...	BloodGroup
Haris	36	1	Iqbal	4	A+
Gaara	38	1	Iqbal	3	A+
Khan	39	1	Iqbal	3	A+
Shahzaib	41	1	Iqbal	10	A+
Haris	43	1	Iqbal	10	A+
Shandana	35	2	Zohaib	2	O+
Zainab	42	2	Zohaib	10	O+
Fahra	37	3	Khadim	2	B-
Umah	40	3	Khadim	3	B-
Waina	44	4	Jazib	3	A+
Wahama	45	4	Jazib	2	A+
Ladain	46	5	Sharjeel	5	B-

Back

## ❖ *bloodrate.java*

Blood Group	Rate per unit
A+	40
A-	40
AB+	20
AB-	3
B+	15
B-	45
O+	4
O-	50

Bloodgroup:

Rate per unit:

## ❖ *payment.java*

Enter Recipient ID:

Amount to be deposited:

Units Purchased:

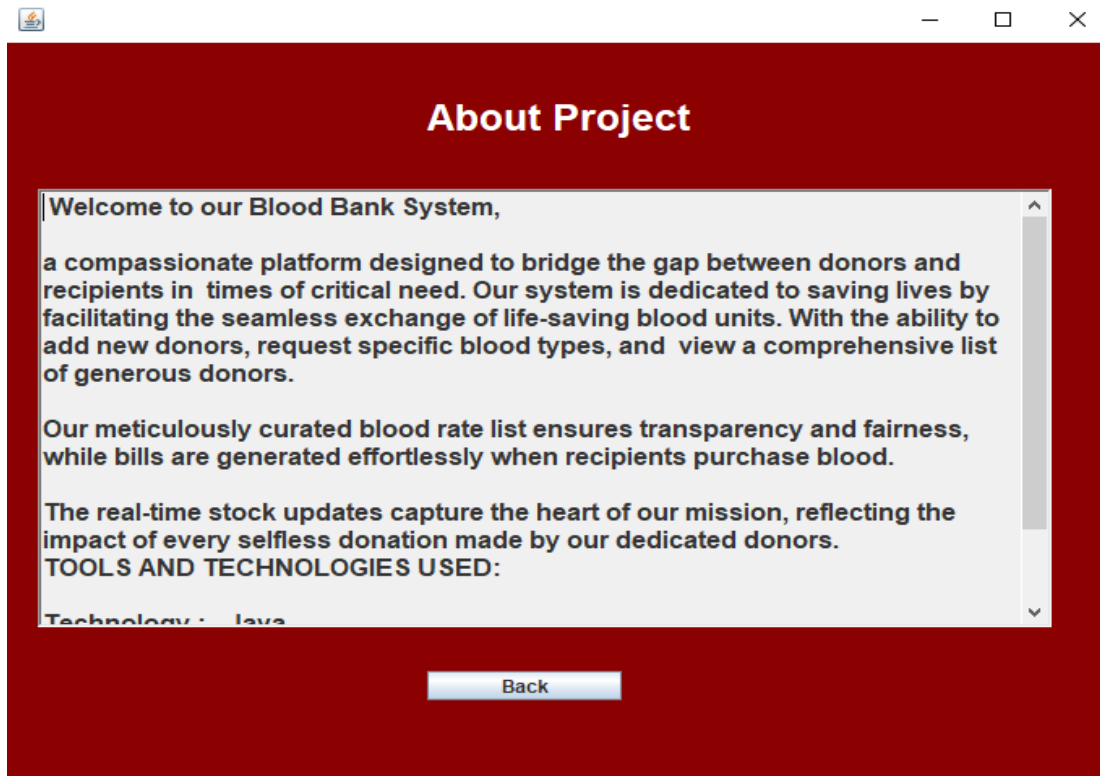
Blood Group:

Bill No:

Name:

Message

Bill Made Successfully.



## ▪ Future Enhancements:

- **Integration with Mobile Platforms:** Develop mobile applications (iOS and Android) to extend the accessibility of the blood bank management system, allowing donors to register, view their donation history, and receive alerts about blood donation drives or urgent blood needs.
- **Enhanced Data Analytics:** Implement advanced analytics capabilities to analyze blood donation trends, predict demand for specific blood types, and optimize blood inventory management. This could involve leveraging machine learning algorithms to forecast blood supply and demand patterns.
- **Telemedicine Integration:** Integrate telemedicine functionalities into the system to facilitate virtual consultations between healthcare professionals and patients requiring blood transfusions. This integration can streamline the process of assessing patient needs and arranging for the appropriate blood products.
- **Blockchain Technology for Blood Supply Chain Management:** Explore the use of blockchain technology to enhance transparency, traceability, and security in the blood supply chain. Blockchain can provide an immutable record of blood donations, transfusions, and inventory movements, reducing the risk of errors or fraudulent activities.



- **Enhanced Donor Engagement:** Implement features to enhance donor engagement and retention, such as personalized donor profiles, gamification elements to encourage regular donations, and social sharing capabilities to raise awareness about blood donation events.
- **Real-time Communication and Alerts:** Introduce real-time communication channels (e.g., SMS, email, push notifications) to notify donors about urgent blood needs, upcoming donation drives, or changes in donation policies. This can help mobilize donors quickly in emergency situations.
- **Biometric Authentication for Donors:** Integrate biometric authentication (e.g., fingerprint or facial recognition) into the system to enhance donor identification and authentication during blood donation processes, ensuring the security and accuracy of donor information.
- **Regulatory Compliance Enhancements:** Stay updated with evolving regulatory requirements in the healthcare industry and incorporate necessary compliance features into the system to ensure adherence to standards such as HIPAA (Health Insurance Portability and Accountability Act) and GDPR (General Data Protection Regulation).
- **Feedback and Quality Improvement Mechanisms:** Implement mechanisms for collecting feedback from donors, recipients, and healthcare professionals to continuously improve the quality of service provided by the blood bank. This feedback loop can drive iterative enhancements to the system based on user input.
- **Enhanced Reporting and Dashboarding:** Develop comprehensive reporting and dashboarding capabilities to provide stakeholders with insights into key metrics such as blood inventory levels, donation trends, recipient demographics, and operational efficiency. Customizable dashboards can empower users to visualize data according to their specific needs.

## ■ Potential Questions:

- How does your system handle the process of generating bills for blood units provided to recipients? Can you explain the steps involved in bill generation and how the system ensures accuracy in calculating the bill amount?
- In your Blood Bank Management System, how are stock levels of blood monitored and displayed to users? Can you explain the process of viewing and checking blood stock, including any alerts or notifications implemented to inform users about low stock levels?
- From an industrial perspective, how does your Blood Bank Management System contribute to addressing critical societal needs such as ensuring timely access to blood donations for medical emergencies and improving the overall efficiency of blood bank operations? Can you discuss any specific features or functionalities that have been instrumental in supporting these objectives?

## ▪ Conclusion:

*In a nutshell*, the development of the Blood Bank Management System represents a significant milestone in modern healthcare technology, addressing the critical need for efficient blood inventory management and donor-recipient coordination. By leveraging Java OOP concepts, MySQL database integration, and user-friendly GUI components, the system offers a robust platform for blood banks to streamline operations, enhance data accuracy, and ultimately save lives. Through its functionalities such as donor management, recipient tracking, billing, and database connectivity, the system empowers healthcare professionals with real-time insights and facilitates seamless coordination between donors, recipients, and blood bank administrators. Moving forward, continuous enhancements and adaptation to emerging technologies will be pivotal in ensuring the system's relevance and effectiveness in serving society's evolving needs. With a commitment to innovation and excellence, the Blood Bank Management System stands as a testament to the power of technology in advancing healthcare delivery and making a tangible difference in communities worldwide.

## ▪ Reference Citations:

- <https://www.javatpoint.com/dbms-tutorial>
- <https://www.freecodecamp.org/news/dbms-and-sql-basics/>
- <https://www.youtube.com/>
- <https://www.w3schools.com/>
- <https://www.codecademy.com/>
- <https://www.learnjavaonline.org/>
- <https://www.simplilearn.com/tutorials/dbms-tutorial>

***THE END***