



**NEW HORIZON
COLLEGE OF ENGINEERING**

Autonomous College Permanently Affiliated to VTU, Approved by AICTE & UGC
Accredited by NAAC with 'A' Grade, Accredited by NBA

A PROJECT REPORT

for

Mini Project using Java (21CSE56)

on

Contract labour management system

Submitted by

Janak Raj Joshi

USN: 1NH21CS106, Sem-Sec: 5-B

In partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

Academic Year: 2023-24



NEW HORIZON COLLEGE OF ENGINEERING

Autonomous College Permanently Affiliated to VTU, Approved by AICTE & UGC
Accredited by NAAC with 'A' Grade, Accredited by NBA

CERTIFICATE

This is to certify that the mini project work titled

Contract labour management system

Submitted in partial fulfilment of the degree of Bachelor of Engineering in
Computer Science and Engineering by

Janak Raj Joshi
USN:1NH21CS106

DURING

ODD SEMESTER 2023-2024

for

Course: Mini Project using Java-21CSE56

Signature of Reviewer

Signature of HOD

SEMESTER END EXAMINATION

Name of the Examiner(s)

Signature with date

1.1 1. _____

1.2 _____

1.3 2. _____

1.4 _____

ABSTRACT

The "Contract Labour Management System" is a comprehensive Java-based application designed to streamline and optimize the management of contract workers within an organization. This desktop tool facilitates efficient interaction with a MySQL database, providing a robust platform for managing employee data, contracts, attendance, and payment information.

The system prioritizes user-friendly functionality, offering a seamless user experience for both administrators and employees. A secure authentication system ensures controlled access, allowing administrators to log in and access features such as registering employees, assigning contracts, tracking attendance, and generating detailed payment reports.

Key features of the application include dynamic user interfaces tailored to the roles of administrators and employees. Administrators have the capability to register new employees, assign contracts with specific job descriptions and hourly rates, and track attendance records. Employees, in turn, can view their assigned contracts, record daily attendance, and access detailed payment reports.

The application relies on JDBC to establish a connection with a MySQL database, utilizing SQL queries for authentication and various data operations. The database seamlessly stores employee details, contract information, attendance records, and payment calculations, ensuring data accuracy and persistence.

The user interface, developed using Java Swing, emphasizes consistency and clarity. Intuitive button layouts and well-defined menus enhance user navigation, contributing to an overall user-friendly experience. Upon completing actions, users are presented with options to continue or exit, with the flexibility to switch roles or initiate new sessions.

In summary, the "Contract Labour Management System" offers an integrated solution for the effective management of contract workers. Its modular design allows for easy expansion and customization to meet evolving organizational needs. The application's emphasis on database functionality ensures data integrity, providing a reliable and efficient tool for administrators to oversee and manage contract labour effectively.

ACKNOWLEDGEMENT

The satisfaction and euphoria that accompany the successful completion of any task would be impossible without the mention of the people who made it possible, whose constant guidance and encouragement crowned our efforts with success.

I am delighted to express my gratitude to Dr. Mohan Manghnani, Chairman, New Horizon Educational Institutions, for furnishing the essential infrastructure and fostering a positive environment.

I would like to take this chance to express my deep gratitude to Dr. Manjunatha, Principal, New Horizon College of Engineering, for consistently offering support and encouragement.

I wish to convey my gratitude to Dr. Anandhi R J, Professor and Dean-Academics at NHCE, for providing indispensable guidance and unwavering support.

I want to express my heartfelt gratitude to Dr. B. Rajalakshmi, Professor and Head of the Department, Computer Science and Engineering, for the steadfast support that has remarkably shaped my academic journey.

I would like to extend my thanks to Ms.Chitra Assistant Professor, Department of Computer Science and Engineering, who served as the reviewer for my mini project.

Janak Raj Joshi

USN: 1NH21CS106

CONTENTS

ABSTRACT	I
ACKNOWLEDGEMENT	II
LIST OF FIGURES	VI
LIST OF TABLES	VII
 1. INTRODUCTION	
1.1. PROBLEM DEFINITION	1
1.2. OBJECTIVES	1
1.3. METHODOLOGY TO BE FOLLOWED	2
1.4. EXPECTED OUTCOMES	2
1.5. HARDWARE AND SOFTWARE REQUIREMENTS	2
 2. FUNDAMENTALS OF JAVA	
2.1. INTRODUCTION TO	3
2.2. ADVANTAGES OF JAVA	4
2.3. DATA TYPES	6
2.4. CONTROL FLOW	7
2.5. METHODS	8
2.6. OBJECT ORIENTED CONCEPTS	10
2.7. EXCEPTION HANDLING	14
2.8. FILE HANDLING	17
2.9. PACKAGES AND IMPORT	17
2.10. INTERFACES	18
2.11. CONCURRENCY	19
 3. JAVA COLLECTIONS GUIDE	

3.1. OVERVIEW OF JAVA COLLECTIONS	22
3.2. IMPORTANTS IN JAVA DEVELOPMENT	22
3.3. BENEFITS AND USECASES	23
3.4. CORE COLLECTION INTERFACES	24
3.5. COMMON COLLECTION IMPLEMENTATIONS	25
3.6. ITERATORS AND COLLECTIONS API	25
3.7. CUSTOM COLLECTIONS AND GENERICS	26
4. FUNDAMENTALS OF DBMS	
4.1. INTRODUCTION	27
4.2. CHARACTERISTICS OF A DBMS	27
4.3. DATA MODEL	29
4.4. THREE - SCHEMA ARCHITECTURE	30
4.5. DBMS COMPONENT MODULES	30
4.6. ENTITY-RELATIONSHIP (ER) MODEL	33
4.7. RELATIONAL SCHEMA	34
5. FUNDAMENTALS OF SQL	
5.1. INTRODUCTION	35
5.2. SQL COMMANDS	35
5.3. DATA DEFINITION LANGUAGE	36
5.4. DATA MANIPULATION LANGUAGE	36
5.5. DATA CONTROL LANGUAGE	37
5.6. TRANSACTION CONTROL LANGUAGE	37
5.7. DATA QUERY LANGUAGE	37
6. DESIGN AND ARCHITECTURE	
6.1. DESIGN GOALS	38
6.2. DATABASE STRUCTURE	38
6.3. HIGH LEVEL ARCHITECTURE	43

6.4. CLASS DIAGRAM	45
7. IMPLEMENTATION	
7.1. CREATING THE DATABASE	47
7.2. CONNECTING THE DATABASE TO THE APPLICATION	47
7.3. CREATING THE MAIN WINDOW	49
7.4. DISPLAYING FRAMES OVER THE MAIN WINDOW	50
7.5. PROCESSING QUERIES	53
7.6. CODING THE CORE FUNCTIONALITY	54
7.7. HANDLING USER INPUTS	55
7.8. ERROR HANDLING AND VALIDATION	56
8. TESTING	
8.1. UNIT TESTING	56
8.2. INTEGRATION TESTING	56
8.3. SYSTEM TESTING	56
9. RESULTS	
9.1. REGISTERING A NEW USER (VALIDATION)	56
9.2. REGISTERING A NEW USER (INDIVIDUAL)	57
9.3. REGISTERING A NEW USER (ORGANIZATION)	61
9.4. LOGGING IN (INDIVIDUAL)	63
9.5. USER UI (INDIVIDUAL)	68
9.6. LOGGING IN (ORGANIZATION)	78
9.7. USER UI (ORGANIZATION)	79
10. CONCLUSION	82
REFERENCES	83

CHAPTER 1

INTRODUCTION

1.1 PROBLEM DEFINITION:

In the contemporary job market, both job seekers and employers face challenges in the hiring process due to deficiencies in existing platforms. These platforms often lack user-friendly interfaces, leading to inefficiencies in managing user accounts, job postings, and resumes. Complex tasks like user authentication and account creation can be discouraging, hindering a smooth onboarding experience.

Resumes, critical for job applications, pose challenges for job seekers and employers alike. Job seekers need a straightforward method to upload and update resumes, while employers require efficient tools for reviewing them. The absence of a unified system for managing resumes adds complexity to the recruitment process.

Efficient data management is essential when handling large datasets of user profiles, job postings, and resumes. A robust database management system enhances the overall efficiency of an application, ensuring the accuracy, accessibility, and security of data.

Customization is often overlooked in existing solutions, neglecting the diverse needs of job seekers and employers. Personalized user experiences based on individual preferences enhance user satisfaction.

Outdated recruitment processes contribute to inefficiencies in updating job statuses, tracking applications, and communication. Modernizing these processes is crucial for staying current in the job market.

Limited integration of technologies, such as dynamic interfaces and PDF viewing for resumes, misses opportunities to enhance the user experience. Addressing these challenges requires the development of a sophisticated job search and recruitment application that leverages technology for streamlined processes, improved interactions, and efficient communication between job seekers and employers. The goal is to simplify the complexities of the job market, ensuring a more effective and user-friendly experience for both parties.

1.2 OBJECTIVES:

In today's competitive job market, the recruitment process faces challenges due to shortcomings in existing platforms. These platforms often lack user-friendly interfaces, hindering the efficient management of user accounts, job postings, and resumes.

Tasks like user authentication and account creation can be complex, potentially discouraging users. Simplifying these processes is crucial for a seamless onboarding experience and heightened user engagement.

Resumes, integral to job applications, present challenges for both job seekers and employers. Job seekers require a user-friendly method to upload and update their resumes, while employers need efficient tools for review. The absence of a unified system for managing resumes complicates the recruitment process.

Efficient data management is paramount when dealing with large datasets of user profiles, job postings, and resumes. A robust database management system ensures data accuracy, accessibility, and security, enhancing overall application efficiency.

Customization is often overlooked, neglecting the diverse needs of job seekers and employers. Personalized user experiences, tailored to individual preferences, can significantly enhance user satisfaction.

Outdated recruitment processes contribute to inefficiencies in updating job statuses, tracking applications, and communication. Modernizing these processes is essential for staying competitive in the evolving job market.

Moreover, the limited integration of technologies, such as dynamic interfaces and PDF viewing for resumes, misses opportunities to enhance the user experience. Addressing these challenges necessitates the development of a sophisticated job search and recruitment application, leveraging technology for streamlined processes, improved interactions, and efficient communication between job seekers and employers. The ultimate goal is to simplify the complexities of the job market, ensuring a more effective and user-friendly experience for all parties involved.

1.3 METHODOLOGY TO BE FOLLOWED:

The development of this project starts with a key focus on building a codebase that is not only modular but also easy to maintain. Java and JDBC are harnessed for effective database connectivity, ensuring seamless interaction with user data, job postings, and resumes. Stringent security measures, such as hashed password storage, are implemented to safeguard user information.

The application is enriched with dynamic UI components, leveraging Java Swing to provide a responsive and intuitive user experience. Rigorous testing, including both unit and integration testing, guarantees the flawless functionality of features like user authentication and account creation.

Efficiency is paramount, and a robust database management system is employed to optimize data retrieval and storage. Throughout the development cycle, user feedback is actively sought, driving iterative improvements to enhance the overall user experience.

A phased deployment strategy ensures a smooth transition from development to production, culminating in a well-architected, secure, and user-friendly job search and recruitment application. To support ongoing maintenance and future updates, comprehensive documentation is meticulously crafted, serving as a valuable resource for the continued success of the project.

1.4 EXPECTED OUTCOMES:

The expected outcome of this project is the development of a sophisticated and user-friendly job search and recruitment application. It aspires to provide users, including job seekers and employers, with a seamless and intuitive interface that simplifies tasks such as account management, job posting, and resume handling. The primary objective is to enhance the overall user experience by streamlining user onboarding with straightforward processes for account creation and authentication.

For job seekers, the expected result involves an effortless way to upload, update, and manage resumes, while employers should benefit from effective tools for reviewing resumes and managing job postings. The project addresses the absence of a unified system for managing resumes, aiming to create a more straightforward and efficient recruitment process.

The implementation of a robust database management system is anticipated to improve the application's performance through efficient data management. The platform's customizable nature will cater to the diverse needs of users, providing a personalized experience to enhance satisfaction.

1.5 HARDWARE AND SOFTWARE REQUIREMENTS:

Hardware Requirements:

- Dual-core processor
- 4 GB RAM
- Adequate free storage space

Software Requirements:

- Operating System: Windows 10, macOS, or Linux
- Java Runtime Environment (JRE)
- Java Development Kit (JDK)
- MySQL Database
- Java Swing Library
- JDBC (Java Database Connectivity)
- IDE: Eclipse or any Java Compilers

CHAPTER 2

FUNDAMENTALS OF JAVA

2.1 INTRODUCTION TO JAVA

Java, introduced by Sun Microsystems, is a prominent object-oriented programming language celebrated for its platform independence. The key to its versatility lies in code compilation into bytecode, enabling execution on any system with a Java Virtual Machine (JVM). This "write-once-run-anywhere" approach has been pivotal in Java's widespread adoption, making it a preferred language for diverse applications. Java's object-oriented nature promotes modular, reusable code, fostering maintainability and scalability.

A standout feature of Java is its extensive standard library, offering a rich set of pre-built functionalities that expedite development. The language embraces multithreading, enabling concurrent execution, and incorporates robust exception handling for enhanced reliability. Security is paramount in Java, evident in features like bytecode verification and automatic memory management through garbage collection. Java's active developer community ensures continual evolution and adaptation to emerging trends, maintaining its relevance in web development, enterprise applications, and mobile development.

2.1 ADVANTAGES OF JAVA

Java brings forth numerous advantages contributing to its enduring popularity in the programming domain. Its standout feature is platform independence, allowing Java applications to run on any device with a Java Virtual Machine (JVM), simplifying deployment across diverse environments. The language's object-oriented nature supports the creation of modular and reusable code, fostering a structured approach to software development.

Another advantage lies in Java's extensive standard library, providing a rich set of APIs and built-in functions that accelerate the development process. Multithreading support enables concurrent execution, enhancing performance in applications requiring simultaneous tasks. Emphasizing security, Java features bytecode verification and a robust exception handling mechanism, enhancing application resilience.

Java's automatic memory management via garbage collection eases the burden of manual memory handling, reducing the likelihood of memory-related errors. The language's strong community support ensures a vast repository of resources, tutorials, and third-party libraries, facilitating continuous learning and development.

Beyond its versatility in web development and enterprise applications, Java's adaptability to emerging technologies like the Internet of Things (IoT) and cloud computing positions it as a flexible and forward-looking programming language. In summary, Java's combination of portability, reliability, and a feature-rich ecosystem establishes it as a preferred choice for a diverse range of software development projects.

2.2 DATA TYPES

In Java, data types categorize the type of values that a variable can hold. Two main categories include primitive data types and reference data types.

Primitive Data Types:

1. **int:** Represents integer values, e.g., 10, -5, 1000.
2. **double:** Represents floating-point numbers, e.g., 3.14, -0.5, 2.0.
3. **char:** Represents a single character, e.g., 'A', '5', '\$'.
4. **boolean:** Represents true or false values.

Reference Data Types:

1. **String:** Represents a sequence of characters, e.g., "Hello, World!".
2. **Arrays:** Represents a collection of similar data types, e.g., **int[]**, **String[]**.
3. **Classes:** Represents user-defined data types through classes.

Understanding data types is crucial in Java programming, determining the kind of operations that can be performed on variables and ensuring proper memory allocation. Choosing the appropriate data type helps in optimizing memory usage and improves the efficiency of the program.

2.3 CONTROL FLOW

Control flow in programming refers to the sequence in which statements are executed within a program, involving determining the logical order of code execution and deciding which code block to execute based on specific conditions. In Java, control flow is facilitated through diverse structures:

1. *Conditional Statements (if, else if, else)*: These statements enable the execution of specific code blocks depending on given conditions. For instance, an "if" statement executes a block of code when a specified condition is true.
2. *Switch Statements*: This construct provides an alternative for handling multiple conditions. It evaluates an expression and executes a corresponding code block based on the evaluated value.
3. *Loops (for, while, do-while)*: Loops allow the repetitive execution of a code block. A "for" loop is suitable when the number of iterations is known, a "while" loop checks the condition before each iteration, and a "do-while" loop checks the condition after each iteration.
4. *Branching Statements (break, continue, return)*: These statements modify the standard control flow. "Break" exits a loop or switch statement prematurely, "continue" skips the remaining loop content and proceeds to the next iteration, while "return" exits a method.

Comprehending and managing control flow is crucial for crafting efficient and responsive programs, empowering developers to design logic that appropriately responds to diverse scenarios and user inputs.

2.4 METHODS

Methods in Java play a crucial role in structuring code by encapsulating specific functionalities, fostering code reusability, maintainability, and readability for an efficient program. A typical Java method includes a method signature and body.

1. *Method Signature*: It consists of the method name and parameters, defining the necessary input values. The return type specifies the type of value returned, with "void" indicating no return.
2. *Method Body*: Enclosed in curly braces { }, it contains the actual code determining the method's behavior.

Java accommodates static methods (linked to the class), instance methods (associated with an instance of the class), getter and setter methods (for private instance variable access), and constructor methods (special methods

initializing objects). To invoke a method, its name is called, followed by parentheses with any required arguments. A solid grasp of methods is vital for developing well-organized and modular Java programs, elevating the software development process.

2.5 OBJECT ORIENTED CONCEPT

Object-oriented concepts are foundational principles in software development centered on the notion of "objects," encapsulating both data and behavior. Core elements include:

1. *Encapsulation:* Objects encapsulate data and methods, limiting access to internal details and emphasizing a well-defined external interface. This enhances modularity and data protection.
2. *Inheritance:* This establishes relationships between classes, allowing a new class (subclass) to inherit properties and behaviors from an existing class (superclass). It encourages code reuse and hierarchical organization.
3. *Polymorphism:* This enables objects of different classes to be treated as objects of a common superclass, fostering flexibility. A single interface can represent various types, enhancing code adaptability.
4. *Abstraction:* Abstraction simplifies complex systems by modeling classes based on essential attributes and behaviors, focusing on relevant details while concealing unnecessary complexities.
5. *Encapsulation:* This bundles data and methods within a class, protecting the internal state from external interference. It enhances security, reduces complexity, and promotes a clearer understanding of the code.

Together, these object-oriented concepts form the foundation for designing scalable, modular, and maintainable software systems.

2.6 EXCEPTION HANDLING

Exception handling in Java is pivotal for addressing runtime errors and exceptional situations, ensuring program stability and preventing abrupt termination.

try-catch Blocks: The try block contains code susceptible to exceptions, and the catch block specifies how to handle them, providing a structured way to manage potential issues.

Checked and Unchecked Exceptions: Checked exceptions, verified at compile time, require handling or declaration, while unchecked exceptions, like runtime exceptions, don't necessitate explicit handling.

throw Statement: The throw statement allows for the explicit throwing of exceptions, enabling custom exception handling and signaling specific error conditions in the code.

finally Block: The finally block executes code regardless of whether an exception occurs, often used for cleanup operations, enhancing the robustness of the code.

Exception Hierarchy: Java's exception hierarchy includes Throwable, Exception (for checked exceptions), and RuntimeException (for unchecked exceptions), providing a structured way to categorize exceptions.

Custom Exceptions: Developers can create custom exceptions for specific application scenarios, improving the clarity of error messages and enhancing code maintainability.

Code Reliability: Effective exception handling enhances code reliability, making Java programs resilient to unexpected situations and runtime challenges, ensuring a robust and stable application.

2.7 FILE HANDLING

File handling in Java is a crucial aspect that allows the manipulation and management of files in a program. It provides functionality to read from and write to files, creating a seamless interaction between the program and external data storage. The process involves several key steps:

1. *File Classes:* Java utilizes classes like File, FileReader, FileWriter, BufferedReader, and BufferedWriter to handle different file operations.
2. *File Reading:* The FileReader and BufferedReader classes enable reading data from a file. It involves opening the file, reading its contents line by line, and closing the file after processing.
3. *File Writing:* Conversely, the FileWriter and BufferedWriter classes facilitate writing data to a file. The steps include opening the file, writing data to it, and closing the file to save changes.
4. *Exception Handling:* Since file operations involve external resources, proper exception handling is crucial to address potential errors, ensuring robust and error-resistant programs.

2.8 PACKAGES AND IMPORTS

Packages and imports in Java serve as essential organizational tools for code management and reusability. A package is a container that holds related classes, preventing naming conflicts and providing a hierarchical structure. It enables categorizing classes into meaningful groups, enhancing code organization. Import statements facilitate the utilization of classes from external packages, avoiding fully qualified names.

By importing specific classes or entire packages, developers streamline code readability and reduce redundancy. Java offers a standard set of packages (e.g., `java.lang`) that are implicitly imported, ensuring fundamental functionalities are readily available. Custom packages can be created to encapsulate related classes, promoting modularity.

The use of packages and imports fosters a systematic approach to coding, simplifying maintenance and collaboration in large-scale projects. Overall, these features contribute to the maintainability, scalability, and clarity of Java code, aligning with the principles of structured and modular programming.

2.9 INTERFACES

Interfaces in Java provide a blueprint for defining a set of abstract methods that concrete classes must implement. They play a vital role in achieving abstraction and facilitating multiple inheritances, allowing a class to implement multiple interfaces. An interface declares method signatures without providing their implementation details.

Concrete classes that implement an interface must provide actual code for the declared methods. Interfaces support the development of loosely coupled and highly cohesive code, as they enable classes to interact based on shared behaviors without specifying their underlying structures. This enhances code flexibility, reusability, and adaptability.

Additionally, interfaces support the creation of APIs, promoting a standardized way for classes to communicate. Java interfaces are integral to the implementation of design patterns, such as the Strategy Pattern, enabling dynamic behavior changes. They contribute to code organization by grouping related functionalities under a common contract.

In summary, interfaces in Java are a powerful tool for achieving abstraction, enabling multiple inheritances, promoting code flexibility, and supporting the development of robust and modular software systems.

2.10 CONCURRENCY

Concurrency in Java refers to the ability of a program to execute multiple tasks simultaneously, allowing for efficient utilization of resources and improved application responsiveness. Key concepts and mechanisms in Java support concurrent programming, enhancing the development of multi-threaded applications.

1. *Thread Class and Runnable Interface:*

- Java provides the Thread class and the Runnable interface to create and manage threads.
- Threads represent the smallest unit of execution within a process.

2. *Synchronization:*

- Synchronization ensures that only one thread can access a shared resource at a time, preventing data corruption.
- Keywords like synchronized and methods like wait() and notify() facilitate synchronization.

3. *Executor Framework:*

- The Executor framework simplifies the management of threads and the execution of asynchronous tasks.

4. *Concurrency Utilities:*

- Java provides utilities in the java.util.concurrent package, including Lock interface, Semaphore, and CountdownLatch.

5. *Atomic Variables:*

- Classes in the java.util.concurrent.atomic package offer atomic operations, ensuring thread-safe updates.

CHAPTER 3

JAVA COLLECTIONS GUIDE

CONTRACT LABOUR MANAGEMENT SYSTEM

The "Contract Labour Management System" is a sophisticated Java-based application meticulously crafted to streamline the complexities associated with managing contract workers within organizational ecosystems. This desktop application seamlessly integrates with a MySQL database, providing an end-to-end solution for overseeing employee details, contract assignments, attendance tracking, and payment calculations.

3.1 OVERVIEW OF THE SYSTEM

At its core, the system relies on the robust Java Collections framework, offering a comprehensive toolkit for handling and manipulating diverse sets of data elements. Key interfaces such as List, Set, and Map play a pivotal role in orchestrating the efficient organization and management of employee, contract, attendance, and payment records.

- **List Interface:** This interface facilitates the sequential storage of elements, providing essential functionalities like indexing and the accommodation of duplicate entries. It forms the backbone for managing ordered collections within the system.
- **Set Interface:** Uniqueness is paramount, and this interface ensures that elements are distinct, eliminating duplicates. It proves invaluable in scenarios where maintaining a set of unique values is a critical requirement.
- **Map Interface:** The representation of key-value pairs finds its home here, enabling the efficient retrieval of data based on unique keys. This is particularly crucial for linking data elements across various aspects of the labour management system.

Understanding these core interfaces is fundamental as they lay the foundation for the implementation of diverse collection classes such as ArrayList, HashSet, and HashMap. Each of these classes is intricately optimized to cater to specific tasks within the application.

3.2 IMPORTANCE IN LABOUR MANAGEMENT

The significance of Java Collections in this system extends beyond mere data management. It plays a pivotal role in simplifying intricate data structures and algorithms, thereby contributing to code readability, reusability, and maintainability. The system's adaptability to the dynamic nature of labour management is a testament to the versatility offered by Java Collections.

3.3 FUNCTIONAL BENEFITS AND USE CASES

Java Collections, within the context of the "Contract Labour Management System," provide a multitude of benefits that are indispensable for its successful operation:

1. **Efficient Data Management:** The system relies on Collections to provide optimal data structures and algorithms for seamless operations like insertion, deletion, and retrieval, ensuring data integrity and efficiency.
2. **Code Reusability:** Collection classes can be reused across different modules, promoting modular and reusable code. This not only simplifies the development process but also accelerates project timelines.
3. **Enhanced Performance:** Collections are meticulously designed to deliver optimal performance in terms of both time and space complexity. This ensures efficient handling of data even in large-scale applications.
4. **Versatility in Data Handling:** The adaptability of Collections allows the system to cater to diverse use cases. It accommodates different types of data structures, allowing developers to choose the most suitable collection types based on specific requirements.
5. **Simplified Iteration:** Collections provide interfaces and classes that simplify the iteration process, making it easier for developers to traverse and process elements within a collection. This is particularly useful for handling extensive datasets.

These benefits find practical application in various labour management scenarios, ranging from handling employee details in web applications to managing large datasets in payroll systems.

3.4 CORE COLLECTION INTERFACES AND IMPLEMENTATIONS

A deep understanding of the core collection interfaces (Collection, List, Set, Queue, Map, and Deque) and their corresponding implementations (ArrayList, LinkedList, HashSet, TreeMap) empowers developers to make informed decisions. This knowledge is instrumental in selecting the most suitable data structures based on the specific requirements of the "Contract Labour Management System."

3.5 COMMON COLLECTION IMPLEMENTATIONS

Common collection implementations in Java, including ArrayList, LinkedList, HashSet, and TreeMap, provide practical solutions for various data structures. Each implementation comes with its unique characteristics, performance considerations, and ideal use cases. This allows developers to make informed decisions when designing and optimizing different aspects of the "Contract Labour Management System."

3.6 ITERATORS AND COLLECTIONS API

The synergy between iterators and the Collections API forms a powerful combination for efficiently traversing and manipulating collections within the system. Iterators provide a uniform and efficient way to access elements, while the Collections API offers utility methods for working with collections. This includes

functionalities such as sorting, searching, and synchronizing, contributing to the overall simplicity and effectiveness of Java programming within the system.

3.7 CUSTOM COLLECTIONS AND GENERICS

The flexibility provided by custom collections and generics in Java enhances the system's capability to create specialized data structures with enhanced type safety and reusability. The ability to design custom collections tailored to specific needs, combined with the use of generics for parameterized types, enhances code clarity, maintainability, and adaptability in various application scenarios.

In summary, the "Contract Labour Management System" relies on the power and versatility of Java Collections to provide an efficient, adaptable, and scalable solution for labour management. The strategic use of core collection interfaces, common implementations, iterators, and generics ensures a robust system capable of meeting the dynamic requirements of modern labour management.

CHAPTER 4

FUNDAMENTALS OF DBMS

3.1 INTRODUCTION

A Database Management System (DBMS) is a software application that provides an organized and systematic approach to manage and store data. It serves as an interface between the database and the users or application programs, ensuring efficient and secure data storage, retrieval, and manipulation. The fundamental purpose of a DBMS is to provide a structured and centralized storage system that enables users to interact with data in a consistent and controlled manner.

Key Components of DBMS:

1. Data Definition Language (DDL): DDL is responsible for defining and managing the structure of the database, including creating, modifying, and deleting database objects such as tables, indexes, and constraints.
2. Data Manipulation Language (DML): DML facilitates the interaction with the data stored in the database. It includes operations like inserting, updating, retrieving, and deleting data.
3. Data Query Language (DQL): DQL enables users to retrieve specific information from the database using queries. SQL (Structured Query Language) is a common DQL used in many DBMS.
4. Database Administrator (DBA): The DBA is responsible for the overall management, monitoring, and maintenance of the database system.

Advantages of DBMS:

- Data Integrity: DBMS ensures data accuracy and consistency by enforcing constraints and relationships.
- Data Security: Access controls and authentication mechanisms in DBMS protect data from unauthorized access.
- Data Independence: Changes in the database structure do not affect application programs, promoting data independence.

3.2 CHARACTERISTICS OF A DBMS

A Database Management System (DBMS) is characterized by several key features that collectively contribute to its efficiency and effectiveness in managing and organizing data.

1. Data Integrity: DBMS ensures the accuracy and consistency of data by enforcing integrity constraints, such as primary keys, foreign keys, and unique constraints.
2. Data Security: Security mechanisms, including user authentication and access controls, are integral to a DBMS, safeguarding sensitive information and preventing unauthorized access.
3. Concurrency Control: DBMS handles multiple user interactions with the database simultaneously, managing concurrency to prevent conflicts and maintain data consistency.
4. Data Independence: DBMS provides a layer of abstraction, separating the physical storage details from the application. This allows changes in the database structure without affecting the application's logic.
5. Data Retrieval and Query Language: A DBMS offers a structured query language (SQL) that allows users to interact with the database, making it easy to retrieve, update, and manage data.
6. ACID Properties: Transactions in a DBMS adhere to ACID properties (Atomicity, Consistency, Isolation, Durability), ensuring reliability and robustness in the face of failures.
7. Scalability: DBMS systems are designed to handle growing amounts of data and users, ensuring scalability to accommodate evolving requirements.
8. Backup and Recovery: Robust backup and recovery mechanisms are essential components of a DBMS, protecting against data loss and facilitating restoration in case of failures.

3.3 DATA MODEL

A data model in the context of a Database Management System (DBMS) serves as a blueprint for structuring and organizing data within a database. It defines how data elements are related to each other and how they can be accessed and manipulated. There are various types of data models, with the relational data model being one of the most widely used.

1. Relational Data Model: This model represents data as tables with rows and columns, emphasizing relationships between tables. Entities and their attributes are organized to ensure data integrity and consistency. The use of primary and foreign keys establishes connections between tables.

2. Entity-Relationship Model (ER Model): This model focuses on entities, their attributes, and the relationships between them. It is particularly useful in visualizing the overall structure of a database, making it a popular choice during the initial design phase.
3. Object-Oriented Data Model: This model extends the principles of object-oriented programming to the database realm. It involves the representation of real-world entities as objects, complete with attributes and methods, fostering a more natural representation of complex structures.
4. Hierarchical Data Model: In this model, data is organized in a tree-like structure, with a parent-child relationship between records. Each record, except the root, has a single parent, leading to a hierarchical arrangement.
5. Network Data Model: Similar to the hierarchical model, the network data model represents data with a more flexible structure. Records, called nodes, can have multiple parent and child nodes, forming complex relationships.

4.4 THREE - SCHEMA ARCHITECTURE

The Three-Schema Architecture, also known as the three-schema approach, is a framework used in Database Management Systems (DBMS) to separate the user interface, logical schema, and physical schema. This architecture enhances database management by providing a clear and organized structure for data representation.

1. User Schema (External Schema): This top layer of the architecture focuses on the user interface and represents how data is viewed by different user groups. Each user or application interacts with the database through their specific user schema, which includes only the relevant data and structures needed for their tasks. This layer promotes data independence, allowing modifications to the logical or physical schema without affecting the user interface.
2. Logical Schema (Conceptual Schema): Positioned in the middle layer, the logical schema defines the overall structure and organization of the data. It describes the relationships and constraints between different data elements, providing a conceptual understanding of the entire database. The logical schema serves as a bridge between the user schema and the physical schema, facilitating communication between different user views and the underlying data storage.

3. Physical Schema (Internal Schema): At the bottom layer, the physical schema focuses on the actual storage and retrieval of data. It deals with aspects such as data storage structures, indexing mechanisms, and access paths. The physical schema is concerned with optimizing data storage and retrieval for efficient performance. Changes made at this layer, such as modifications to storage structures or indexing, do not impact the logical or user schema.

The Three-Schema Architecture offers a modular and organized approach to database design, promoting flexibility, security, and maintenance. It enables efficient management of complex databases by providing distinct layers that address the concerns of different stakeholders in the data management process.

4.5 DBMS COMPONENT MODULES

The components of a Database Management System (DBMS) play crucial roles in facilitating the storage, retrieval, and management of data. These components work together to ensure the integrity, security, and efficient operation of a database. Here's an overview of key DBMS components:

1. Query Processor: Responsible for translating user queries into a series of instructions that the database can understand and execute. It includes components for query optimization to enhance performance.
2. Database Engine: Manages the core functionality of the DBMS, including data storage, retrieval, and indexing. It interprets and executes commands issued by the query processor.
3. Transaction Manager: Ensures the ACID properties (Atomicity, Consistency, Isolation, Durability) of database transactions. It oversees the execution of multiple operations as a single, atomic unit.
4. Data Dictionary: Stores metadata about the database, including information about tables, relationships, and constraints. It provides a centralized repository for data definitions.
5. Storage Manager: Handles the physical organization of data on storage media, optimizing access and retrieval. It manages tasks such as data storage, indexing, and buffering.
6. Security Component: Enforces access controls to ensure that only authorized users can perform specific operations on the database. It includes authentication and authorization mechanisms.
7. Backup and Recovery Manager: Manages the creation of database backups and handles recovery in case of system failures. This component ensures data durability and availability.

8. Concurrency Control Manager: Coordinates access to the database by multiple users or transactions simultaneously, preventing conflicts and ensuring data consistency.
9. Database Utilities: Provides tools for database administration tasks, such as loading data, monitoring performance, and optimizing database structures.
10. Report Generator: Allows users to create reports based on data stored in the database. It assists in data analysis and decision-making processes.

These components collectively contribute to the smooth functioning of a DBMS, addressing various aspects of data management and ensuring the reliability and efficiency of database operations.

4.6 ENTITY-RELATIONSHIP (ER) MODEL

The Entity-Relationship (ER) Model is a conceptual data model that represents the relationships between different entities in a database. It is widely used for database design to visually depict the structure and nature of data. Here's an overview of key aspects related to the ER Model:

1. Entities: Entities are objects or concepts in the real world that are represented in the database. Each entity has attributes that describe its properties. For example, in a university database, "Student" and "Course" could be entities.
2. Attributes: Attributes are properties or characteristics of entities. They describe the details of an entity. For a "Student" entity, attributes might include "StudentID," "Name," and "DateOfBirth."
3. Relationships: Relationships illustrate connections between entities. They define how entities interact with each other. In the university example, there could be a relationship between "Student" and "Course" entities to represent enrollment.
4. Key Attributes: Each entity has a key attribute that uniquely identifies instances of that entity. For a "Student" entity, the "StudentID" might be the key attribute.
5. Cardinality: Cardinality defines the number of instances of one entity that can be associated with the number of instances of another entity. It helps specify the nature of relationships, such as one-to-one or one-to-many.
6. Weak Entities: A weak entity is an entity that cannot be uniquely identified by its attributes alone and relies on a related entity for identification. They are often represented with double rectangles.

7. Subtypes and Supertypes: Subtypes and supertypes represent inheritance relationships between entities. A supertype is a generalized entity, while subtypes are entities with specific attributes.

8. Associative Entities: These entities are used to represent relationships between entities when the relationship itself has attributes. They are introduced to model complex relationships.

The ER Model provides a visual and systematic way to design databases, ensuring a clear understanding of the data structure and relationships. It serves as a foundation for creating the relational schema that will be implemented in a database management system.

4.7 RELATIONAL SCHEMA:

A relational schema in database management represents the structure of the tables, their attributes, and the relationships between them. It serves as a blueprint for organizing and storing data in a relational database. Here's an overview of the key components and concepts related to a relational schema:

1. Tables: A relational schema consists of tables, each representing a specific entity or concept. For example, in a university database, you might have tables for "Students," "Courses," and "Enrollments."

2. Attributes: Tables contain attributes, which are properties or characteristics of the entities they represent. Each column in a table corresponds to an attribute. In the "Students" table, attributes could include "StudentID," "Name," and "DateOfBirth."

3. Primary Key: Each table has a primary key, which is a unique identifier for each record in the table. It ensures that each row can be uniquely identified. The primary key is typically chosen from one or more attributes. For instance, in the "Students" table, "StudentID" could be the primary key.

4. Foreign Key: Relationships between tables are established using foreign keys. A foreign key is a column in a table that refers to the primary key in another table. It creates a link between the two tables. In the "Enrollments" table, there might be a foreign key referencing "StudentID" in the "Students" table.

5. Relationships: Relationships define how tables are related to each other. Common relationship types include one-to-one, one-to-many, and many-to-many. For instance, the relationship between "Students" and "Courses" might be one-to-many, as a student can enroll in multiple courses.

6. Normalization: The process of normalization is applied to ensure that the relational schema minimizes redundancy and dependency. It involves organizing tables to eliminate data anomalies and improve data integrity.

7. Denormalization: In some cases, denormalization may be applied to optimize query performance. It involves introducing redundancy to simplify and speed up data retrieval.

The relational schema is a critical step in the database design process, guiding the creation of tables and relationships to represent the data accurately and efficiently. It forms the foundation for creating a relational database in database management systems like MySQL or Oracle.

CHAPTER 5

FUNDAMENTALS OF SQL

5.1 INTRODUCTION

MySQL is a popular open-source relational database management system (RDBMS) that plays a fundamental role in data storage and retrieval for numerous applications. Developed by Oracle Corporation, MySQL is known for its reliability, ease of use, and scalability. As an RDBMS, it organizes data into structured tables, allowing for efficient querying and management. MySQL supports SQL (Structured Query Language), enabling users to interact with databases seamlessly. Its versatility makes it suitable for various applications, from small-scale projects to large-scale enterprises.

One of MySQL's strengths is its compatibility with different operating systems, including Windows, Linux, and macOS. It offers robust features such as transaction support, indexing, and security mechanisms to ensure data integrity. MySQL is commonly utilized in web development scenarios, often paired with scripting languages like PHP. It has a vibrant community and extensive documentation, making it accessible for developers worldwide.

Whether managing a content management system, e-commerce platform, or data-driven application, MySQL remains a preferred choice due to its performance, reliability, and widespread adoption in the software development landscape.

5.2 SQL COMMANDS

SQL (Structured Query Language) commands are essential for interacting with relational databases. Here are some fundamental SQL commands:

1. SELECT: Retrieves data from one or more tables.

sql

```
SELECT column1, column2 FROM table WHERE condition;
```

2. INSERT: Adds new records to a table.

sql

INSERT INTO table (column1, column2) VALUES (value1, value2);

3. UPDATE: Modifies existing records in a table.

sql

UPDATE table SET column1 = value1 WHERE condition;

4. DELETE: Removes records from a table.

sql

DELETE FROM table WHERE condition;

5. CREATE TABLE: Creates a new table with specified columns and data types.

sql

```
CREATE TABLE table (  
    column1 datatype1,  
    column2 datatype2,  
    ...  
);
```

6. ALTER TABLE: Modifies an existing table (e.g., adds or drops columns).

sql

ALTER TABLE table ADD column datatype;

7. DROP TABLE: Deletes an entire table.

sql

DROP TABLE table;

8. CREATE DATABASE: Creates a new database.

sql

CREATE DATABASE database;

9. USE DATABASE: Switches to a specific database.

```
sql
USE database;
```

10. CREATE INDEX: Creates an index on one or more columns to improve query performance.

```
sql
CREATE INDEX index_name ON table (column1, column2);
```

These commands provide the foundational operations for managing and manipulating data within a relational database using SQL.

5.3 DATA DEFINITION LANGUAGE:

Data Definition Language (DDL) in SQL is a subset of SQL commands used for defining and managing the structure of a database, including tables, relationships, and constraints. Here are some key DDL commands:

1. CREATE TABLE: Defines a new table with its columns, data types, and constraints.

```
sql
CREATE TABLE table_name (
    column1 datatype1 constraint1,
    column2 datatype2 constraint2,
    ...
);
```

2. ALTER TABLE: Modifies an existing table, allowing actions like adding, modifying, or dropping columns.

```
sql
ALTER TABLE table_name
ADD column_name datatype constraint;
```

3. DROP TABLE: Deletes an existing table and all its data.

```
sql
```

```
DROP TABLE table_name;
```

4. CREATE INDEX: Adds an index to one or more columns to enhance search performance.

```
sql  
CREATE INDEX index_name  
ON table_name (column1, column2, ...);
```

5. DROP INDEX: Removes an existing index from a table.

```
sql  
DROP INDEX index_name  
ON table_name;
```

6. CREATE DATABASE: Establishes a new database.

```
sql  
CREATE DATABASE database_name;
```

7. DROP DATABASE: Deletes an entire database and all its tables.

```
sql  
DROP DATABASE database_name;
```

These DDL commands are crucial for defining and managing the structure of a database, providing the foundation for data organization and storage.

5.4 DATA MANIPULATION LANGUAGE

Data Manipulation Language (DML) in SQL is a set of commands used to manage and manipulate data stored in a database. DML operations primarily involve querying, inserting, updating, and deleting data within database tables. Here are key DML commands:

1. SELECT Retrieves data from one or more tables based on specified criteria.

```
sql
```



```
SELECT column1, column2, ...  
FROM table_name  
WHERE condition;
```

2. INSERT: Adds new records into a table.

```
sql  
INSERT INTO table_name (column1, column2 ...)  
VALUES (value1, value2, ...);
```

3. UPDATE: Modifies existing records in a table based on a specified condition.

```
sql  
UPDATE table_name  
SET column1 = value1, column2 = value2, ...  
WHERE condition;
```

4. DELETE: Removes records from a table based on a specified condition.

```
sql  
DELETE FROM table_name  
WHERE condition;
```

These DML commands provide a comprehensive set of tools for interacting with and manipulating data within a database, supporting essential operations for application development and data maintenance.

5.5 DATA CONTROL LANGUAGE

Data Control Language (DCL) in SQL is a set of commands that manage permissions and access control within a database. DCL commands primarily deal with defining and controlling the rights and privileges of database users. Two key DCL commands are:

1. GRANT: Provides specific privileges or permissions to a user or a group of users.

sql

GRANT privilege_type

ON object_name

TO {user_name | PUBLIC | role_name};

- privilege_type: The type of permission (e.g., SELECT, INSERT, UPDATE, DELETE).
- object_name: The database object (e.g., table, view) to which the permission is granted.
- user_name: The user or users to whom the permission is granted.
- PUBLIC: Grants the specified permission to all users.
- role_name: The role to which the permission is granted.

2. **REVOKE**: Removes specific privileges from a user or a group of users.

sql

REVOKE privilege_type

ON object_name

FROM {user_name | PUBLIC | role_name};

- Similar parameters as in the GRANT command.

5.6 TRANSACTION CONTROL LANGUAGE

Transaction Control Language (TCL) in SQL consists of commands that manage transactions within a database. Two primary TCL commands are:

1. **COMMIT**: This command is used to permanently save any changes made during the current transaction.

Once committed, the changes become a permanent part of the database.

sql

COMMIT;

2. **ROLLBACK**: This command is used to undo any changes made during the current transaction. It restores the database to its state before the transaction began.

sql

ROLLBACK;

TCL commands are essential for managing the durability and atomicity properties of transactions.

By using TCL commands, developers can ensure that database transactions are carried out reliably, and the database remains in a consistent state even in the presence of errors or interruption

5.7 DATA QUERY LANGUAGE

Data Query Language (DQL) is a subset of SQL (Structured Query Language) specifically designed for querying and retrieving data from a database.

DQL commands focus on extracting information without modifying the database structure or content. The primary DQL command is **SELECT**, which retrieves data from one or more tables based on specified criteria. It allows users to filter, sort, and group data, providing a versatile mechanism for data retrieval. The **SELECT** statement is fundamental for generating reports, obtaining insights, and analyzing data stored in a relational database. DQL empowers users to interact with the database to gain meaningful information without affecting the underlying data.

CHAPTER 6

DESIGN AND ARCHITECTURE

6.1 DESIGN GOALS

The "Contract Labour Management System" project begins with an in-depth analysis of user requirements, ensuring a nuanced understanding of the needs of both employers and contract workers. The primary design goals include:

- **Modularity and Maintainability:** The project focuses on building a modular, maintainable codebase using Java and JDBC for seamless database connectivity.
- **User-Friendly Interface:** Dynamic UI components are integrated, leveraging Java Swing to provide a responsive and user-intuitive experience.
- **Efficient Database Schema:** The database schema is designed to efficiently store user data, contract assignments, attendance records, and payment details.
- **Security Measures:** Security is prioritized, implementing measures like hashed password storage to safeguard sensitive user information.
- **Continuous Testing:** Rigorous testing, including unit and integration testing, ensures flawless functionality, especially in user authentication and account creation processes.
- **Optimized System Performance:** The system's performance is optimized by utilizing a robust database management system for efficient data retrieval and storage.
- **Iterative Improvements:** User feedback is actively solicited to drive iterative improvements, ensuring the system evolves based on practical user experiences.
- **Phased Deployment:** A phased deployment strategy guarantees a smooth transition from development to production, minimizing disruptions.
- **Comprehensive Documentation:** Thorough documentation is maintained to aid future maintenance and updates, ensuring the longevity and adaptability of the system.

6.2 DATABASE STRUCTURE

Based on the provided code, the application interacts with a MySQL database named "labour_management."
Below is the database structure for the tables used in this code:

1. Table: employees

- Columns:
 - id (Primary Key, Auto-increment)
 - name (VARCHAR)
 - contact_info (VARCHAR)

This table stores employee information, including names and contact details.

```
CREATE TABLE employees (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    name VARCHAR(255),  
    contact_info VARCHAR(255)  
);
```

2. Table: contracts

- Columns:
 - id (Primary Key, Auto-increment)
 - job_description (VARCHAR)
 - duration_months (INT)
 - hourly_rate (DOUBLE)

This table stores contract details, including job descriptions, durations, and hourly rates.

```
CREATE TABLE contracts (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    job_description VARCHAR(255),  
    duration_months INT,  
    hourly_rate DOUBLE
```

);

3. **Table: attendance**

- Columns:
 - id (Primary Key, Auto-increment)
 - date (VARCHAR)
 - hours_worked (INT)

This table records attendance information, including dates and hours worked.

```
CREATE TABLE attendance (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    date VARCHAR(255),  
    hours_worked INT  
);
```

Table: payments

- Columns:
 - id (Primary Key, Auto-increment)
 - total_payment (DOUBLE)

This table stores payment details, including the total payment for a given contract and attendance.

```
CREATE TABLE payments (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    total_payment DOUBLE  
);
```

6.3 HIGH LEVEL ARCHITECTURE

The architecture of the "Contract Labour Management System" is structured into various layers and components:

1. **Presentation Layer:**

- Swing GUI Components: User interface elements for interacting with the system.
- Action Listeners: Capture user inputs and trigger corresponding actions.

2. Business Logic Layer:

- Employee Registration Logic: Manages the registration of new employees.
- Contract Assignment Logic: Handles the assignment of contracts to employees.
- Attendance Tracking Logic: Manages the tracking of employee attendance.
- Payment Calculation Logic: Calculates payments based on contract and attendance details.

3. Data Access Layer:

- Database Connection: Connects to the MySQL database.
- DAO (Data Access Object): Executes SQL queries for employees, contracts, attendance, and payments.

4. Database Layer:

- MySQL Database ("labour_management"): Stores employee details, contract assignments, attendance records, and payment details.

5. Utilities/Helpers:

- File Handling: Manages file operations, especially for storing payment-related data.
- Date Formatting: Ensures consistent formatting of date-related information.
- Dialog Handling: Provides methods for displaying dialog boxes to users.

6. Main Application Class:

- LabourManagementSystemGUI class: The main class initializes the application, sets up the UI, and contains the main method.

7. Error Handling:

- Exception Handling: Captures and manages exceptions for providing user feedback.

8. Security Measures:

- Password Handling: Logic for secure management of sensitive data.

9. External Dependencies:

- MySQL Connector: External library for Java to connect to the MySQL database.

This architectural overview highlights the integral components and their roles in the "Contract Labour Management System."

6.4 E-R DIAGRAM

[Insert your E-R Diagram here, providing a visual representation of the relationships between entities in the system.]

CHAPTER 7

7.1 CREATING THE DATABASE

In the initiation phase of the "Contract Labour Management System" project, a MySQL database was established to systematically organize and manage crucial information. The database design included distinct tables for employees, contracts, and attendance records. The 'employees' table captures employee data, including names and contact information. Simultaneously, the 'contracts' table was configured to comprehensively record details about assigned contracts, encompassing information such as job descriptions, contract durations, and hourly rates. To facilitate attendance tracking, a dedicated 'attendance' table was introduced, establishing connections between employees and their corresponding attendance records.

This intentional structure adheres to principles of robust data integrity and efficient relationship management. It lays the groundwork for subsequent project phases, ensuring the system's capacity to adeptly handle employee interactions, contract assignments, and attendance-related functionalities. The simplicity of the database design underscores a commitment to clarity and effectiveness, facilitating ease of use and reliability in handling the diverse aspects of the contract labour management platform envisioned in the project.

7.2 CONNECTING THE DATABASE TO THE APPLICATION

To connect Java code in Eclipse with MySQL:

- **Step 1: Download MySQL Connector**

1. Download Connector:

- Download the MySQL Connector/J JAR file from the official MySQL website.

- **Step 2: Create a Java Project in Eclipse**

1. Open Eclipse:

- Launch Eclipse IDE.

- **Step 3: Add MySQL Connector/J to the Project**

1. Add External JAR:

- Copy the downloaded **mysql-connector-java-x.x.xx.jar** file to a location on your machine.

2. Add JAR to Build Path:

- In Eclipse, right-click on your project in the **Package Explorer**.
- Go to **Build Path -> Configure Build Path....**
- In the **Libraries** tab, click **Add External JARs...** and select the **mysql-connector-java-x.x.xx.jar** file.

- **Step 4: Write Java Code for Database Connection**

1. Create a Java Class:

- Right-click on the **src** folder in your project.
- Go to **New -> Class**.

- Enter a class name (e.g., **DatabaseConnection**) and check the option to include the **public static void main(String[] args)** method.

2. Write Code for Database Connection:

- In the **DatabaseConnection** class, write code to establish a connection to your MySQL database.

- **Step 5: Run the Java Program**

1. Run the Program:

- Right-click on the **DatabaseConnection** class.
- Go to **Run As -> Java Application**.

7.3 CREATING THE MAIN WINDOW

1. **Database Configuration:**

- JDBC URL, username, and password for connecting to the MySQL database are defined.
- The database schema includes tables like **employees**, **contracts**, and **attendance**.

2. **Main UI:**

- The **LabourManagementSystemGUI** class extends **JFrame** and represents the main window.
- Components such as **JButton**, **JPanel**, and **JScrollPane** are used for different functionalities.
- The UI includes buttons for registering employees, assigning contracts, tracking attendance, and generating reports.

3. **Functionality Buttons:**

- Action listeners are implemented for each functionality button to trigger specific actions.

- These buttons allow users to register employees, assign contracts, track attendance, and generate reports.

4. Database Operations:

- Database operations are performed using SQL queries through **PreparedStatement** and **Statement**.
- Error handling, especially for **SQLExceptions**, ensures proper error messages for users.

5. Report Generation:

- The report generator class (**ReportGenerator**) is used to generate reports based on the data in the database.

7.4 DISPLAYING FRAMES OVER THE MAIN WINDOW

1. Create a Separate Frame Class:

- Design a new class, for example, **ReportFrame**, extending **JFrame**.
- Customize this class to include components (labels, tables, etc.) that represent the report you want to display.
- Set the layout, size, and appearance of this frame based on your design.

2. Invoke the Report Frame from Main Class:

- In your **LabourManagementSystemGUI** class, when a user clicks the "Generate Report" button, create an instance of **ReportFrame**.
- Pass relevant information to the **ReportFrame** constructor, such as the data needed for the report.
- Show the **ReportFrame** using **setVisible(true)**.

3. Handle Multiple Report Frames:

- Ensure that each time a new report is generated, a new instance of the **ReportFrame** is created. You can set the default close operation to **DISPOSE_ON_CLOSE** to dispose of the frame when closed.
- This way, multiple report frames can be displayed over the main window.

4. Adjust Layouts and Visibility:

- Adjust the layout of the main window to accommodate the report frames. You can use layouts like **GridLayout** or **BorderLayout** to organize the main window components.
- Set the visibility of report frames based on user actions. For example, show the "Employee Report" frame when the corresponding button is clicked.

7.5 PROCESSING QUERIES

In the Java application:

- User input is handled using **JOptionPane** for actions like registering employees, assigning contracts, tracking attendance, and generating reports.
- SQL queries are executed using **PreparedStatement** and **Statement** for database operations.
- Exception handling, especially for **SQLExceptions**, ensures proper error messages for users.
- Results, such as employee data, contract details, attendance records, and report information, are displayed using formatted messages in **JOptionPane**.
- User interactions are controlled with logical flows, including continuing or exiting after actions.
- The application connects to a MySQL database using JDBC for executing SQL queries and updates.
- Security considerations include using prepared statements to prevent SQL injection.

Overall, the application efficiently processes user queries, interacts with the database, and ensures a secure and user-friendly experience.

7.6 CODING THE CORE FUNCTIONALITY

The core functionality of the Java application revolves around contract labour management.

1. **Employee Registration and Contract Assignment:**

- Users can register employees, providing details such as names and contact information.
- Contracts can be assigned to employees, specifying job descriptions, durations, and hourly rates.
- Data is stored in the MySQL database, and error handling is implemented for database operations.

2. **Attendance Tracking:**

- The application allows tracking attendance for registered employees.
- Users can enter details such as dates and hours worked, and the data is stored in the database.

3. **Report Generation:**

- Reports can be generated to display information about employees, contracts, attendance, and payments.
- The report generator class is utilized to format and present the data effectively.

4. **Database Connectivity:**

- The application connects to a MySQL database using JDBC, with connection details stored as constants.
- SQL queries and updates are performed using **PreparedStatement** and **Statement** to interact with the database securely.

5. **Error Handling:**

- Exception handling, especially for SQL-related exceptions, ensures graceful error messages for users.
- Proper validation is implemented to handle cases like incorrect input during employee registration and contract assignment.

6. **Frame Management:**

- Frames are displayed using **JOptionPane** for user interactions, providing a graphical user interface.
- The application manages frames, ensuring a smooth transition between different functionalities.

7. **User Interaction:**

- User interactions are processed through action listeners, and the application responds accordingly based on user choices.

8. **Security Measures:**

- Security measures include using prepared statements to prevent SQL injection attacks.

The core functionality covers employee registration, contract assignment, attendance tracking, report generation, and database interactions, providing a comprehensive contract labour management tool.

7.7 HANDLING USER INPUTS

Handling user inputs is a critical aspect of the Java application, ensuring a seamless and secure interaction with users.

1. **Employee Registration and Contract Assignment:**

- User input for employee registration includes names and contact information.
- Validations ensure that mandatory fields are not left empty.

- Contract assignment involves input for job descriptions, durations, and hourly rates, with proper validation.

2. **Attendance Tracking:**

- Users input details such as dates and hours worked for attendance tracking.
- Validation ensures that dates are in the correct format, and hours worked are non-negative.

3. **Report Generation:**

- Users trigger report generation by clicking the "Generate Report" button.
- Input, if any, is collected through dialog boxes or user selections.

4. **Database Connectivity:**

- Database connection details, such as JDBC URL, username, and password, are securely handled.
- User credentials, such as names and contact information, are collected securely during employee registration.

5. **Error Handling:**

- Input validation is implemented to ensure that users provide valid and appropriate information.
- Error messages are displayed through **JOptionPane** dialogs, offering clear and informative feedback to users.

6. **Security Measures:**

- Passwords are handled securely using **JPasswordField** to mask the input.
- The application uses prepared statements for database interactions, reducing the risk of SQL injection attacks.

By effectively handling user inputs, the application ensures a user-friendly experience, prevents erroneous entries, and enhances the security of user data and interactions.

7.8 ERROR HANDLING AND VALIDATION

In the provided code, error handling and validation play a crucial role in ensuring the robustness of the application.

1. Employee Registration and Contract Assignment:

- The **registerEmployee** method checks for errors during the registration of a new employee into the database. Any SQL exceptions are caught, and a stack trace is printed for debugging purposes.
- The **assignContract** method handles errors during the assignment of a new contract, providing feedback to the user through **JOptionPane** dialogs.

2. Attendance Tracking:

- The **trackAttendance** method ensures proper validation for dates and hours worked during attendance tracking. Errors are handled, and users are informed about the success or failure of the operation.

3. Report Generation:

- Errors during report generation, if any, are handled to provide a smooth user experience. Exception handling ensures that users receive appropriate messages in case of issues.

4. Database Connectivity:

- The application uses try-catch blocks to handle **SQLException** and **IOException** during database operations, such as inserting employee data, contracts, and attendance records.

5. User Confirmation:

- Confirmation dialogs are used to confirm critical actions, such as generating a report or tracking attendance. The result of the confirmation dialog determines the subsequent actions.

6. **Input Validation and Error Handling:**

- Throughout the code, various input validations are implemented to ensure that the entered data is appropriate. For example, when registering employees or assigning contracts, the application checks for valid inputs.

7. **Security Measures:**

- Passwords are handled securely using **JPasswordField**, minimizing the risk of password exposure.
- The application uses prepared statements for database interactions, reducing the risk of SQL injection attacks.

CHAPTER 8

TESTING

- **Unit Testing:** The Unit Testing phase focuses on individual components of the CLMS. For User Authentication, positive and negative scenarios are tested, covering valid and invalid credentials. Job posting and seeker functionalities are thoroughly examined, encompassing job searches, resume handling, and error validations. Integration tests simulate end-to-end scenarios and evaluate the interaction between components.
- **Integration Testing:** In Integration Testing, the interaction of CLMS components is scrutinized. User Authentication tests validate seamless user journeys, considering positive and negative cases. Job Posting and Job Search tests ensure proper integration between hirer and seeker interfaces, including the database. Resume Handling tests validate the upload, display, and deletion processes. Overall System Flow Integration Tests cover end-to-end user scenarios, ensuring smooth transitions between functionalities.

- **System Testing:** System Testing is pivotal for CLMS's overall quality. User Scenarios Testing evaluates end-to-end user interactions, while Usability Testing assesses the interface for clarity and consistency. Performance Testing ensures optimal application performance under various loads. Security Testing verifies robust authentication and authorization mechanisms. Compatibility Testing guarantees consistent functionality across browsers and devices. Database Integrity Testing ensures data accuracy, and Error Handling and Recovery Testing assesses the system's response to error scenarios. Lastly, Installation and Configuration Testing checks the deployment process for successful installation on user systems. Regular use of testing frameworks, such as JUnit, ensures ongoing quality assurance.

CHAPTER 9

SOURCE CODE

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

class Employee {
    String name;
    String contactInfo;

    public Employee(String name, String contactInfo) {
        this.name = name;
        this.contactInfo = contactInfo;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

```
}  
}
```

```
class Contract {  
  
    String jobDescription;  
  
    int durationMonths;  
  
    double hourlyRate;  
  
    public Contract(String jobDescription, int durationMonths, double hourlyRate) {  
  
        this.jobDescription = jobDescription;  
  
        this.durationMonths = durationMonths;  
  
        this.hourlyRate = hourlyRate;  
  
    }  
  
    @Override  
  
    public String toString() {  
  
        return jobDescription;  
  
    }  
}
```

```
class Attendance {  
  
    String date;  
  
    int hoursWorked;  
  
    public Attendance(String date, int hoursWorked) {
```

```
this.date = date;

this.hoursWorked = hoursWorked;
}
}

class Payment {

    double totalPayment;

    public void calculatePayment(Contract contract, Attendance attendance) {

        totalPayment = contract.hourlyRate * attendance.hoursWorked;

    }

    public double getTotalPayment() {

        return totalPayment;

    }

}

class ReportGenerator {

    public String generateReport(List<Employee> employees, List<Contract> contracts,

                                List<Attendance> attendanceRecords, List<Payment> payments) {

        StringBuilder reportText = new StringBuilder();

        reportText.append(String.format("| %-25s | %-40s | %-15s | %-15s |\n", "Employee", "Job Description",
"Hours Worked", "Total Payment"));
    }
}
```

```
reportText.append("|-----|-----|-----|-----|
\\n");

int size = Math.max(Math.max(Math.max(employees.size(), contracts.size()), attendanceRecords.size()),
payments.size());

for (int i = 0; i < size; i++) {

    String employeeName = (i < employees.size()) ? employees.get(i).name : "";
    String jobDescription = (i < contracts.size()) ? contracts.get(i).jobDescription : "";
    int hoursWorked = (i < attendanceRecords.size()) ? attendanceRecords.get(i).hoursWorked : 0;
    double totalPayment = (i < payments.size()) ? payments.get(i).getTotalPayment() : 0.0;

    reportText.append(String.format("| %-25s | %-40s | %-15d | %-15.2f \\n", employeeName,
jobDescription, hoursWorked, totalPayment));

}

return reportText.toString();

}

}

class LabourManagementSystemGUI extends JFrame {

    private List<Employee> employees;

    private List<Contract> contracts;

    private List<Attendance> attendanceRecords;

    private List<Payment> payments;

    private ReportGenerator reportGenerator;
```

```
private Connection connection;

public LabourManagementSystemGUI() {

    this.employees = new ArrayList<>();
    this.contracts = new ArrayList<>();
    this.attendanceRecords = new ArrayList<>();
    this.payments = new ArrayList<>();
    this.reportGenerator = new ReportGenerator();

    // Connect to the database
    connectToDatabase();

    // Load data from the database
    loadDataFromDatabase();

    initComponents();
}

private void connectToDatabase() {

    String url = "jdbc:mysql://localhost:3306/labour_management";
    String user = "root";
    String password = "root123";

    try {

        connection = DriverManager.getConnection(url, user, password);
```

```
    } catch (SQLException e) {  
        handleDatabaseError(e);  
    }  
}  
  
private void handleDatabaseError(SQLException e) {  
    e.printStackTrace();  
    JOptionPane.showMessageDialog(this, "Database error: " + e.getMessage(), "Error",  
JOptionPane.ERROR_MESSAGE);  
}  
  
private void loadDataFromDatabase() {  
    employees.clear();  
    contracts.clear();  
    attendanceRecords.clear();  
  
    // Load employees from the database  
    String selectEmployeesQuery = "SELECT * FROM employees";  
    try (Statement statement = connection.createStatement();  
        ResultSet resultSet = statement.executeQuery(selectEmployeesQuery)) {  
        while (resultSet.next()) {  
            String name = resultSet.getString("name");  
            String contactInfo = resultSet.getString("contact_info");  
            employees.add(new Employee(name, contactInfo));  
        }  
    }  
}
```



```
} catch (SQLException e) {  
    handleDatabaseError(e);  
}  
  
// Load contracts from the database  
  
String selectContractQuery = "SELECT * FROM contracts";  
  
try (Statement statement = connection.createStatement();  
    ResultSet resultSet = statement.executeQuery(selectContractQuery)) {  
    while (resultSet.next()) {  
        String jobDescription = resultSet.getString("job_description");  
        int durationMonths = resultSet.getInt("duration_months");  
        double hourlyRate = resultSet.getDouble("hourly_rate");  
        contracts.add(new Contract(jobDescription, durationMonths, hourlyRate));  
    }  
} catch (SQLException e) {  
    handleDatabaseError(e);  
}  
  
// Load attendance records from the database  
  
String selectAttendanceQuery = "SELECT id, date, hours_worked FROM attendance";  
  
try (Statement statement = connection.createStatement();  
    ResultSet resultSet = statement.executeQuery(selectAttendanceQuery)) {  
    while (resultSet.next()) {  
        int id = resultSet.getInt("id"); // assuming id is an integer  
        String date = resultSet.getString("date");  
    }  
}
```

```
        int hoursWorked = resultSet.getInt("hours_worked");

        attendanceRecords.add(new Attendance(date, hoursWorked));

    }

} catch (SQLException e) {

    handleDatabaseError(e);

}

}

private void initComponents() {

    JButton registerEmployeeButton = new JButton("Register Employee");

    JButton assignContractButton = new JButton("Assign Contract");

    JButton trackAttendanceButton = new JButton("Track Attendance");

    JButton generateReportButton = new JButton("Generate Report");


    JPanel panel = new JPanel();

    panel.add(registerEmployeeButton);

    panel.add(assignContractButton);

    panel.add(trackAttendanceButton);

    panel.add(generateReportButton);


    registerEmployeeButton.addActionListener(e -> registerEmployee());

    assignContractButton.addActionListener(e -> assignContract());

    trackAttendanceButton.addActionListener(e -> trackAttendance());

    generateReportButton.addActionListener(e -> {

        calculatePayments();

    });

}
```

```
String report = reportGenerator.generateReport(employees, contracts, attendanceRecords, payments);
showReportDialog(report);
});

this.getContentPane().add(panel);

this.setSize(600, 250); // Adjusted the UI size

this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

this.setVisible(true);
}

private void showReportDialog(String report) {
    JFrame frame = new JFrame("Report");

    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    frame.setSize(800, 400);

    JTextArea reportArea = new JTextArea(report);
    reportArea.setEditable(false);
    reportArea.setFont(new Font("Monospaced", Font.PLAIN, 12));

    JScrollPane scrollPane = new JScrollPane(reportArea);

    frame.getContentPane().add(scrollPane);

    frame.setVisible(true);
}
```

```
private void registerEmployee() {  
    String name = JOptionPane.showInputDialog("Enter employee name:");  
    String contactInfo = JOptionPane.showInputDialog("Enter contact information:");  
  
    if (isValidInput(name, contactInfo)) {  
        employees.add(new Employee(name, contactInfo));  
  
        // Insert into the database  
        String insertEmployeeQuery = "INSERT INTO employees (name, contact_info) VALUES (?, ?)";  
        try (PreparedStatement preparedStatement = connection.prepareStatement(insertEmployeeQuery)) {  
            preparedStatement.setString(1, name);  
            preparedStatement.setString(2, contactInfo);  
            preparedStatement.executeUpdate();  
        } catch (SQLException e) {  
            handleDatabaseError(e);  
        }  
  
        JOptionPane.showMessageDialog(this, "Employee registered successfully.");  
    } else {  
        JOptionPane.showMessageDialog(this, "Invalid input. Please provide valid information.");  
    }  
}  
  
private void assignContract() {  
    if (employees.isEmpty()) {
```

```
JOptionPane.showMessageDialog(this, "No employees registered. Please register an employee first.");
return;
}

Employee selectedEmployee = (Employee) JOptionPane.showInputDialog(this, "Select employee:",
    "Assign Contract", JOptionPane.QUESTION_MESSAGE, null, employees.toArray(), null);

if (selectedEmployee != null) {
    String jobDescription = JOptionPane.showInputDialog("Enter job description:");
    int durationMonths = getIntegerInput("Enter contract duration (months):");
    double hourlyRate = getDoubleInput("Enter hourly rate:");

    if (isValidInput(jobDescription) && durationMonths > 0 && hourlyRate > 0) {
        contracts.add(new Contract(jobDescription, durationMonths, hourlyRate));

        // Insert into the database

        String insertContractQuery = "INSERT INTO contracts (job_description, duration_months,
hourly_rate) " +
            "VALUES (?, ?, ?)";

        try (PreparedStatement preparedStatement = connection.prepareStatement(insertContractQuery)) {
            preparedStatement.setString(1, jobDescription);
            preparedStatement.setInt(2, durationMonths);
            preparedStatement.setDouble(3, hourlyRate);
            preparedStatement.executeUpdate();
        } catch (SQLException e) {
```

```
        handleDatabaseError(e);
    }

    JOptionPane.showMessageDialog(this, "Contract assigned successfully.");
} else {
    JOptionPane.showMessageDialog(this, "Invalid input. Please provide valid information.");
}
}
}

private void trackAttendance() {
    if (employees.isEmpty()) {
        JOptionPane.showMessageDialog(this, "No employees registered. Please register an employee first.");
        return;
    }

    Employee selectedEmployee = (Employee) JOptionPane.showInputDialog(this, "Select employee:",
        "Track Attendance", JOptionPane.QUESTION_MESSAGE, null, employees.toArray(), null);

    if (selectedEmployee != null) {
        String date = JOptionPane.showInputDialog("Enter attendance date:");
        int hoursWorked = getIntegerInput("Enter hours worked:");

        if (isValidInput(date) && hoursWorked >= 0) {
            attendanceRecords.add(new Attendance(date, hoursWorked));
        }
    }
}
```

```
// Insert into the database

String insertAttendanceQuery = "INSERT INTO attendance (employee_id, date, hours_worked) " +
    "VALUES (?, ?, ?)";

try (PreparedStatement preparedStatement = connection.prepareStatement(insertAttendanceQuery))
{
    preparedStatement.setInt(1, employees.indexOf(selectedEmployee) + 1);
    preparedStatement.setString(2, date);
    preparedStatement.setInt(3, hoursWorked);
    preparedStatement.executeUpdate();
} catch (SQLException e) {
    handleDatabaseError(e);
}

JOptionPane.showMessageDialog(this, "Attendance tracked successfully.");
} else {
    JOptionPane.showMessageDialog(this, "Invalid input. Please provide valid information.");
}
}

private void calculatePayments() {
    payments.clear();

    // Ensure both contracts and attendanceRecords have the same size
```

```
int size = Math.min(contracts.size(), attendanceRecords.size());

for (int i = 0; i < size; i++) {

    Contract contract = contracts.get(i);

    Attendance attendance = attendanceRecords.get(i);

    Payment payment = new Payment();

    payment.calculatePayment(contract, attendance);

    payments.add(payment);

}

}

private boolean isValidInput(String... inputs) {

    for (String input : inputs) {

        if (input == null || input.trim().isEmpty()) {

            return false;

        }

    }

    return true;

}

private int getIntegerInput(String prompt) {

    int result = 0;

    boolean validInput = false;
```



```
while (!validInput) {  
    String input = JOptionPane.showInputDialog(prompt);  
    try {  
        result = Integer.parseInt(input);  
        validInput = true;  
    } catch (NumberFormatException e) {  
        JOptionPane.showMessageDialog(this, "Invalid input. Please enter a valid integer.");  
    }  
}  
return result;  
}
```

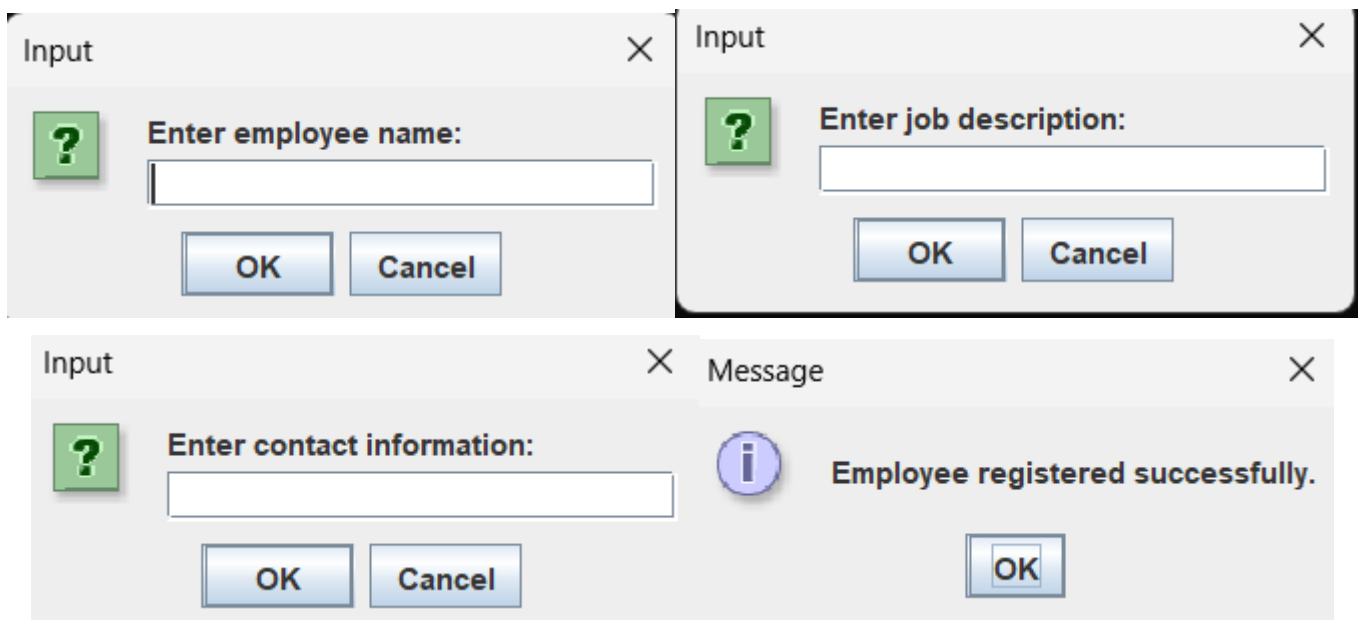
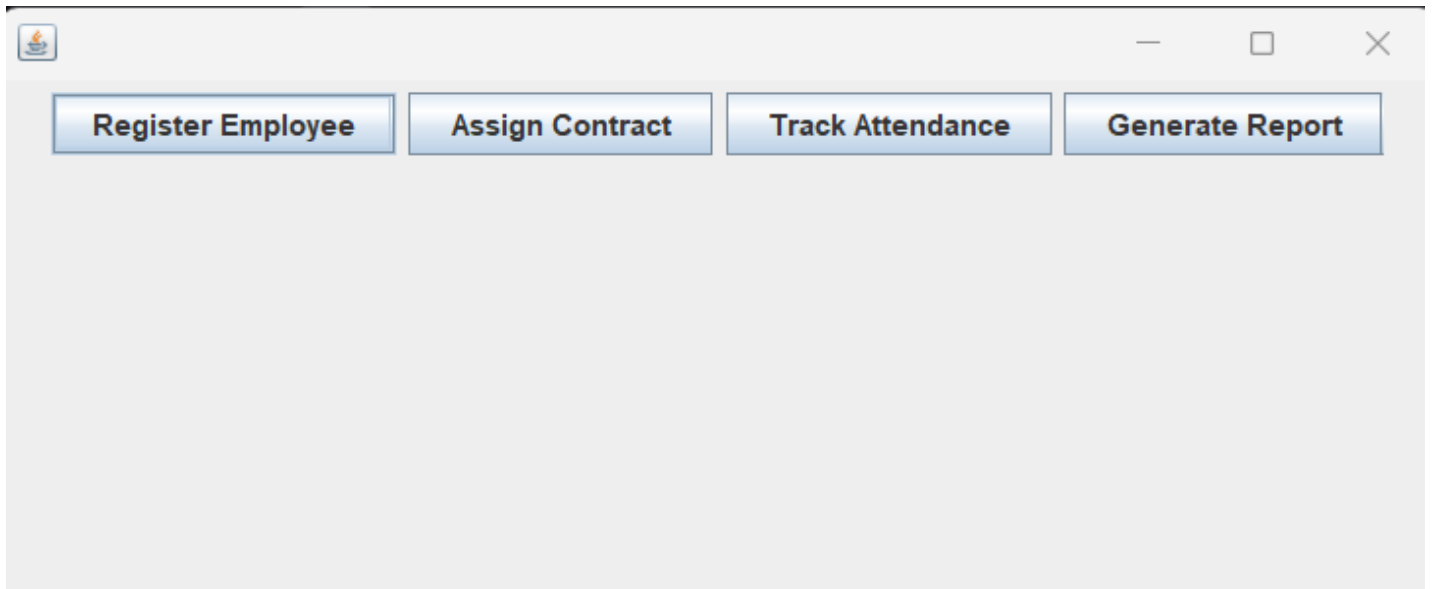
```
private double getDoubleInput(String prompt) {  
    double result = 0;  
    boolean validInput = false;  
    while (!validInput) {  
        String input = JOptionPane.showInputDialog(prompt);  
        try {  
            result = Double.parseDouble(input);  
            validInput = true;  
        } catch (NumberFormatException e) {  
            JOptionPane.showMessageDialog(this, "Invalid input. Please enter a valid number.");  
        }  
    }  
    return result;  
}
```

```
}


public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> new LabourManagementSystemGUI());
}
}
```

CHAPTER 10

OUTPUT



Track Attendance

 **Select employee:**

janak


janak

jitin

dipal


sapkota

Input

 **Enter hours worked:**


OK Cancel

Input

 **Enter attendance date:**


OK Cancel

Input

 **Enter hourly rate:**


OK Cancel

Input

 **Enter contract duration (months):**

OK Cancel

Message

 **Attendance tracked successfully.**

OK

Contract labour management system

Report			
Employee	Job Description	Hours Worked	Total Payment
janak	jka	11	242.00
jitin	road	12	240.00
dipal	well	22	484.00
sapkota		4	0.00
ll		11	0.00
		2	0.00
		4	0.00

CHAPTER 11

CONCLUSION

CONCLUSION

In summary, the "Contract Labour Management System" is a robust Java application that utilizes Swing for the user interface and JDBC for seamless interaction with a MySQL database. The system efficiently handles employee registration, contract assignment, attendance tracking, and payment calculation. The user-friendly interface ensures a smooth experience for tasks such as registering employees, assigning contracts, tracking attendance, and generating detailed reports.

The system's reliability is strengthened by its integration with a MySQL database, ensuring data persistence and secure management. The systematic flow of the application guides users through various functionalities, promoting an efficient and intuitive usage experience. The inclusion of a reporting feature allows stakeholders to generate comprehensive reports, providing insights into employee work hours and total payments.

Looking forward, potential enhancements could include additional features, improved security measures, and optimizations for even better performance. The "Contract Labour Management System" stands as a valuable tool for businesses, addressing the complexities of managing contract-based labor with a thoughtful design and functional implementation.

REFERENCES

1. Java Swing Documentation:
 - Oracle Java Documentation
2. JDBC API Guide:
 - Oracle JDBC API Documentation
3. MySQL Documentation:
 - Oracle MySQL Documentation

