codefinity

JDBC Overview

codefinity

Databases, DBMS

Prerequisites

- Java Basics
- Java Extended
- Java OOP
- Java Data Structures
- Java JUnit Library. Types of Testing
- Multithreading in Java
- Introduction to SQL
- Intermediate SQL
- Relational Database and Normalization

Databases

Welcome to the course! This is another step towards **becoming a Java Developer**, keep it up!

This course will teach you how to **work with databases** within Java. From the course **Relational Database and Normalization** you should already know **what a database is**, the different **types of databases**, and how to use them correctly. However, let's quickly review the **fundamental concepts** once again.

Let's start from the basics, what is a database:

> **Study more**
>
> A database is a **structured collection of data**, typically stored and accessed electronically from a **computer system**. It allows for the efficient organization, storage, retrieval, and manipulation of data. Databases are foundational in **various applications**, ranging from websites, business systems to scientific research, enabling data handling in a systematic and coherent manner.

Great, you should also be aware that **databases are managed through** Database Management Systems (**DBMS**). We will also be using a DBMS called **MySQL** in this course. In order to complete the assignments in this course, you will need to install this DBMS on your device.

> **Note**
>
> It's good if you already know how to use other DBMS like Postgres, MongoDB, or SQL Server, but in this course, we will specifically use MySQL. So, it's preferable to use MySQL for the course, as if you encounter issues with other DBMS, neither I nor the community will be able to assist you :).

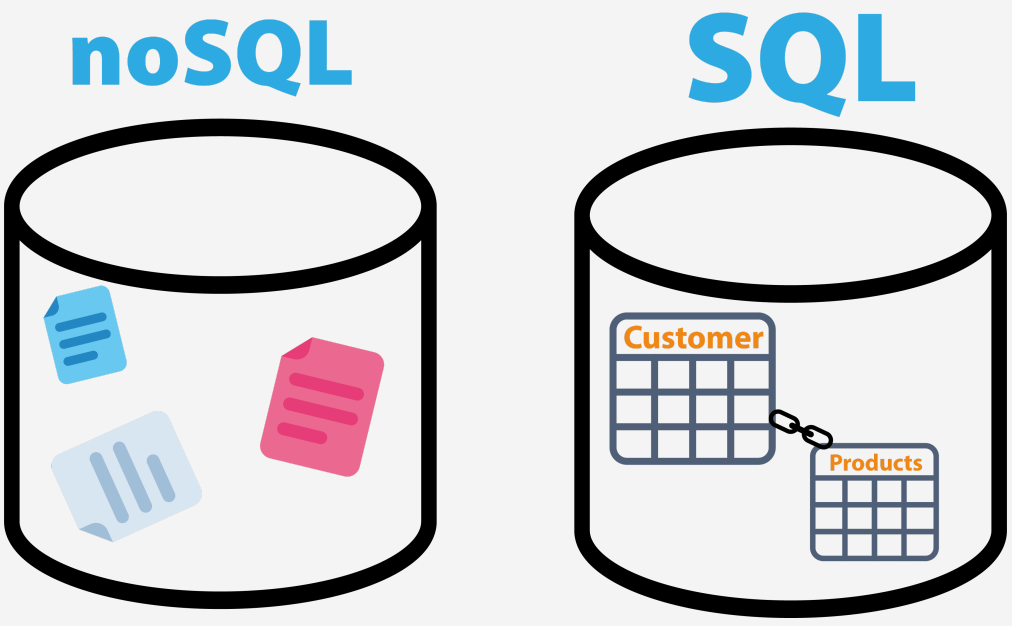
If you happen to encounter any **issues with installing MySQL**, you can refer to the step-by-step installation guide in this **article**. (clickable)

Types of Databases

There are **two** main types of databases that are in constant competition with each other: relational (**SQL**) and non-relational (**NoSQL**) databases.

- **Relational Databases (RDBMS)**: This is the more common type of database. Data is **organized** in the form of **tables with columns and rows**. They use the Structured Query Language (SQL) to store, modify, and retrieve data. Examples of RDBMS include MySQL, Microsoft SQL Server, and PostgreSQL;

- **Non-Relational Databases (NoSQL)**: These databases use a **less structured** way of storing data. They are suitable for **large datasets** where data access **speed and scalability** are of particular importance. Examples of NoSQL databases include MongoDB, Cassandra, and Redis.



SQL

Since we will be working with a **relational database**, it's important to understand what SQL is and what it's used for. We will use SQL frequently to create and manipulate the databases we work with. In case you've forgotten, here's a **brief reminder**:

codefinity

Study more

SQL (Structured Query Language) is a **standardized programming language** used for managing and manipulating **relational databases**. It allows for storing, retrieving, updating, and deleting database records and is essential for tasks such as querying data, setting database schemas, and controlling access to the database's data.

With SQL, we can perform four fundamental operations for working with databases, represented by the acronym **CRUD**:

- **Create** involves entering new data, adding new rows to a table, or creating new documents;
- **Read** includes selecting and reading existing data;
- **Update** involves modifying data in existing records or objects;
- **Delete** includes removing existing records or objects from the database.

You can learn more about CRUD operations in this chapter. (clickable)

Basic SQL Operations

Let's go over the **main SQL operations** that we will use in this course. You should already know these operations from the **SQL from Zero to Hero** track.

- `SHOW DATABASES` displays all available databases:

```
SHOW DATABASES;
```

- `USE` specifies which database to use for executing queries:

```
USE products;
```

- `SHOW TABLES` displays information about tables:

```
SHOW TABLES;
```

- `CREATE TABLE` creates new tables:

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    position VARCHAR(50),
    salary DECIMAL(10, 2),
    hire_date DATE
);
```

- `SELECT` is used to retrieve data from a table:

```
SELECT * FROM employees
```

- `INSERT` adds new records to a table:

```
INSERT INTO employees (id, name, position, salary) VALUES
(1, 'John Doe', 'Software Engineer', 60000.00),
(2, 'Jane Smith', 'Project Manager', 65000.00),
(3, 'Jim Brown', 'Designer', 55000.00),
(4, 'Jake Blues', 'System Analyst', 70000.00);
```

- `UPDATE` modifies existing records in a table:

```
UPDATE employees SET name = 'Peter Griffin' WHERE id = 1
```

- `DELETE` deletes records from a table:

```SQL DELETE FROM employees WHERE id = 2 ```

These are the **main operations** you need to be aware of.

> **Note**
>
> Please note that operations should be written in **uppercase**.

The basic material about databases has been **reviewed and absorbed**. Now, we can move on to more advanced usage of databases in **Java-based** web applications!
```

codefinity

📖 **JDBC Introduction**

How Applications Interact with Databases

Obviously, **merely telling the compiler** that we will be using an external database is not enough to connect to a database. We need tools for that.

In programming, code constantly interacts with **various components**, such as a server or a database; code is just a **big set of instructions** for a device.

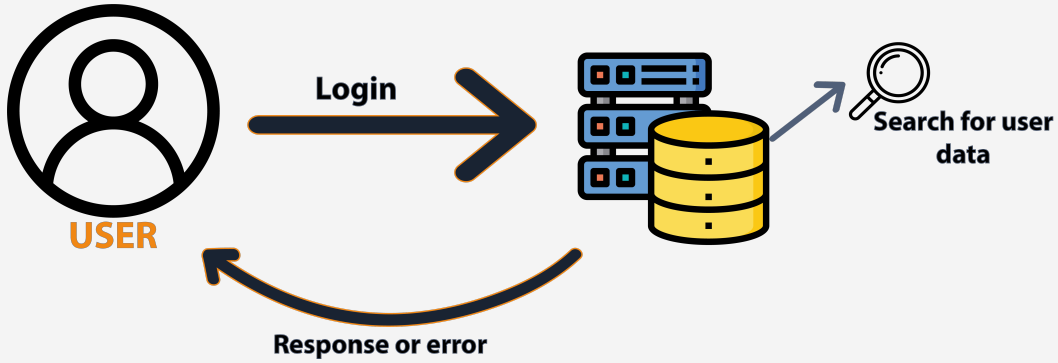
For example, let's say a new user has **registered on your website**, and you need to **store that user's data** somewhere so they can log in and use the product again. There are two ways to do this:

1. **Save the user's data locally on their device in RAM**, similar to how we store data in a variable. However, there's a slight catch: the user can use their profile **until their device is rebooted or the RAM is cleared**. After that, all **information** about their profile **will be lost**, including any purchases they made;

2. **Save the data on the server in a database:** This is exactly what **we'll be learning** in this course. All user data or different products' information is stored in the database. When a user registers, their **data is recorded** in the database. Then, when the user logs in, the **code sends a request to the server's database**, asking it to check if that user exists. If the **response is positive**, the user logs in under their profile and continues working. If the server cannot find the user's data in the database, it will **display an error message**, suggesting that the user may not exist.

Let's look at a visualization of such interaction. First, a user **registers** on our website, and their data goes into the database:

Then, when the user **logs in to the platform again**, the database will look for the user's data, after which they can either log in or receive an error:



I think we can draw a conclusion from all of this: **Everyone uses databases**, storing user data and product data in them. It's **reliable, fast, and convenient**.

But let's answer the **main question** of this course: how does Java connect to a database?

The **answer** is JDBC.

JDBC

JDBC stands for *Java Database Connectivity*. It is an **API** that allows developers to create Java applications and **connect them to databases**.

codefinity

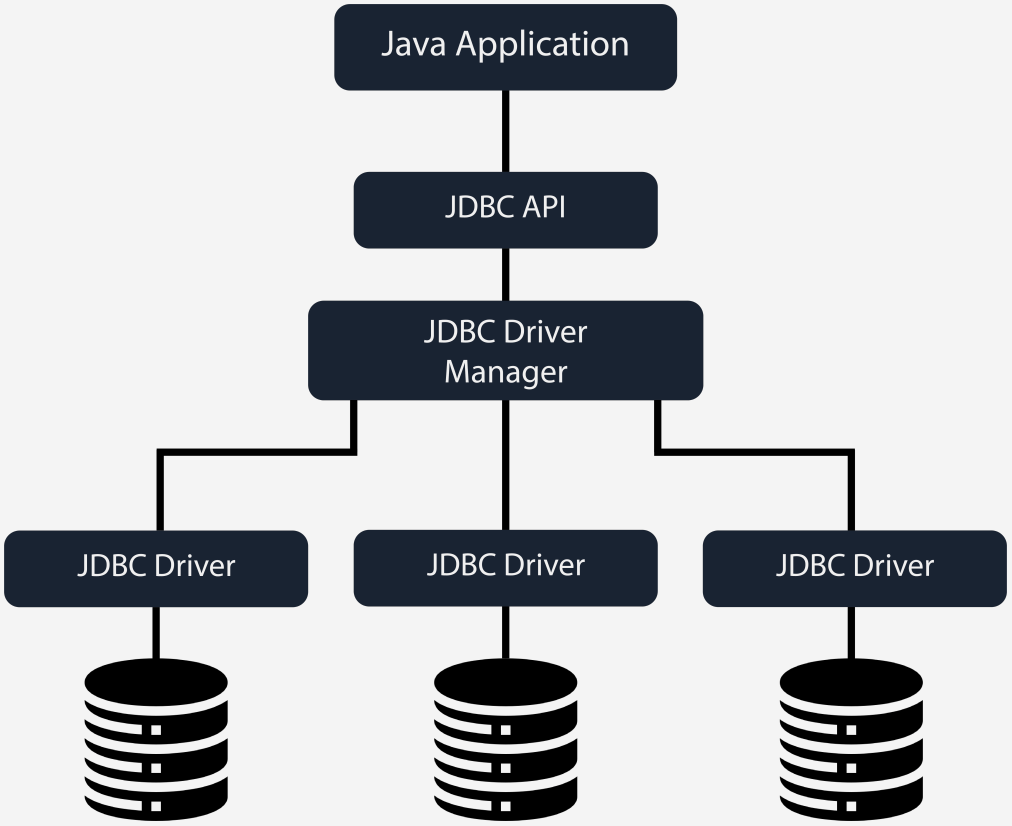
Study more

API (*Application Programming Interface*) in programming is a set of rules, protocols, and tools for **building software and applications**. It defines the methods by which different software components interact with each other, allowing separate parts of a computer program to communicate. An API can be designed for web services, operating systems, databases, or other software applications, facilitating the **development and integration of software products**.

In simple terms, **JDBC is a tool** that allows us to **connect our program to a database**.

> **Note**
>
> It's worth noting that JDBC is **used all the time**; it's a low-level aspect of programming. Even the ORM framework Hibernate, which we will use to create and modify databases in this course, **uses JDBC**.

JDBC has its own **structure and hierarchy**; let's take a closer look at it:



Let's take a closer look at what's happening in this diagram:

- The **JDBC API** provides a link between the application and JDBC management;
- A **JDBC Driver Manager** facilitates the **interaction** between JDBC and the database driver;
- A **JDBC Driver** is a component that implements the JDBC interface for a **specific DBMS**. Each DBMS has its own driver. The JDBC Driver Manager is responsible for **managing database drivers** and establishing connections between the Java application and a specific database.

Conclusion

We can conclude that the **JDBC** working principle is **not as complex as it may seem**; in fact, it resembles the operation of a typical IT company. A Java application **contacts** the JDBC API, requesting **communication with a database**. At the same time, the JDBC API reaches out to the JDBC Driver Manager to **find the appropriate driver for database communication**. The JDBC Driver Manager excels at its job and **selects the necessary driver for a specific DBMS**, after which we connect our code to that database.

JDBC also includes **various components**, which serve as instructions for their use in code. We will discuss these components and their usage in the next chapter!

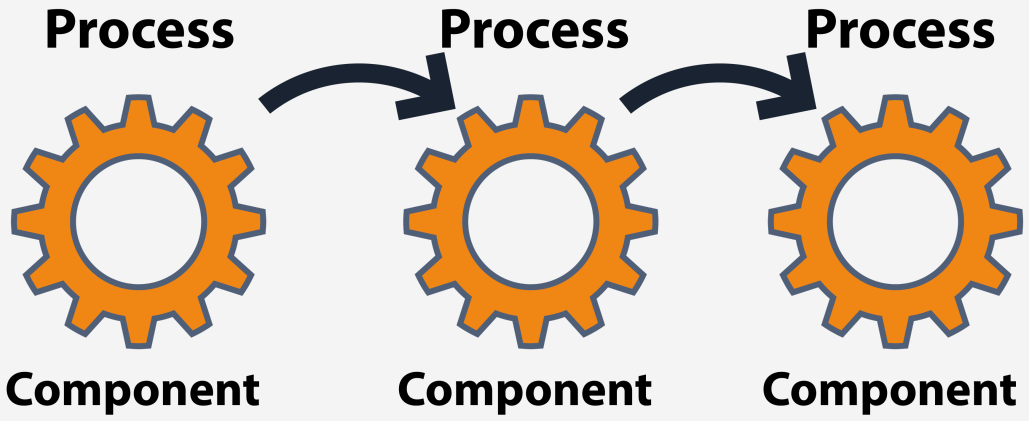
Components of JDBC

Every API is **composed of components**, each responsible for its part of the API's functionality. This structure can also be likened to a **production line**, where each employee has a role in each process.

For instance, we can logically **divide the processes into several sub-processes:**

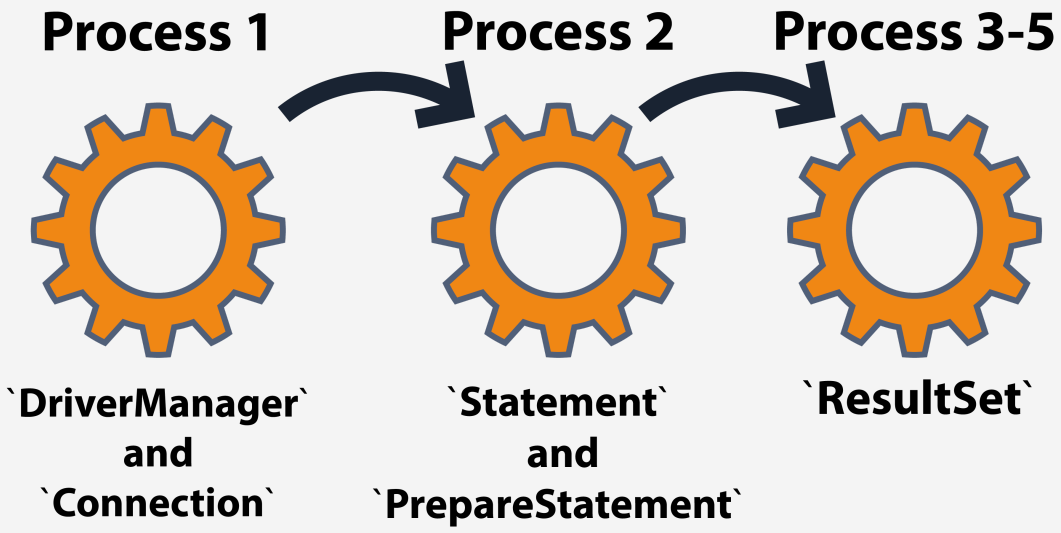
1. **Establish a connection** to the database;
2. **Send an SQL query** to the database;
3. **Receive data** in the form of a response;
4. **Properly interpret this data**, adapting it to Java code;
5. **Transform the data** into a more convenient data structure (e.g., a list, collection, or map).

For each of these processes, there is a corresponding component responsible for successfully executing its instructions. Let's get acquainted with these components in more detail:



- **Connection:** This interface establishes a connection to the database. You use it to create instances of `Statement` or `PreparedStatement`;
- **Statement and PreparedStatement:** These interfaces are used to **execute SQL queries**. `Statement` is used for executing **static** SQL queries without parameters, while `PreparedStatement` is designed for executing SQL queries with **one or more parameters**;
- **ResultSet:** When you execute an SQL query to retrieve data (`SELECT`), the `ResultSet` stores the retrieved data, allowing you to **process each row of the result** sequentially;
- **DriverManager:** This class manages a list of **available JDBC drivers**. It selects the appropriate driver when establishing a connection to the database.

So, from this, we can draw a conclusion about how responsibilities are distributed in the code:



- The `DriverManager` and `Connection` classes perform the **first process**: Establish a connection to the database. The `DriverManager` **selects the appropriate driver** for the specific DBMS, and the `Connection` uses this driver to **connect the code to the database**;
- `Statement` and `PreparedStatement` handle the **second process**, as they allow us to **send SQL queries** to the database and instruct the database on what data we want to retrieve;
- `ResultSet` handles the **remaining processes**. It is with the help of this class and its methods that we can obtain the data in a format convenient for us, such as a `List`.

Thus, we can see that each of the components serves its purpose. This structure is correct and adheres to **SOLID** principles.

> **Note**
>
> We will discuss **SOLID** principles in a separate course. These principles define **well-written application code** by providing guidelines for writing code correctly.

JDBC Interaction with a Database

Let's examine the **interaction** of JDBC with a database **through code** examples.

It may seem complicated at first, but as you progress through the course, especially when working on practical assignments, you will gain a better understanding of how it works and what needs to be done.

To begin with, let's see what the database we will be working with and connecting to looks like. We will be working with the `testDatabase` schema and the `employees` table:

id	name	position	salary	hire_date
1	John Doe	Software Engineer	60000.00	2021-01-10
2	Jane Smith	Project Manager	65000.00	2021-02-15
3	Jim Brown	Designer	55000.00	2021-03-01
4	Jake Blues	System Analyst	70000.00	2021-04-20

A fairly simple table in the database. Now, let's set ourselves the **task** of retrieving data such as `id`, `name`, and `salary` from the first row in the table (**id=1**):

```java
package codefinity;

import java.sql.*;

public class HibernateExample {
    static String jdbcUrl = "jdbc:mysql://localhost:3306/testDatabase";
    static String username = "db_username";
    static String password = "db_password";
    public static void main(String[] args) {

        try {
            Connection connection = DriverManager.getConnection(jdbcUrl
            PreparedStatement statement = connection.prepareStatement("
            ResultSet resultSet = statement.executeQuery();
            if (resultSet.next()) {
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                double salary = resultSet.getDouble("salary");
                System.out.println("ID: " + id + ", Name: " + name + ",
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

> **Note**
>
> You can find instructions on how to properly configure MySQL and retrieve the `db_username` and `db_password` here: **article**.

Let's go through it step by step:

Step	Action	Description
1	**Variable Creation**	Create `String` variables to store the **database URL** (which includes the database name, equivalent to SQL's `USE DatabaseName`), the MySQL `username`, and the user's `password`. Details are hidden for privacy.
2	**Connection Establishment**	Use the `Connection` interface and the `DriverManager` class to establish a database connection. This involves calling the `getConnection()` method with the database URL, username, and password as parameters, thereby establishing a connection for sending SQL queries.
3	**Preparing Statement**	Employ the `prepareStatement()` method on the connection object, and assign the resulting prepared statement object to a variable. This object contains the SQL query specified in the method's parameter.
4	**Executing Query**	Utilize the `executeQuery()` method on the prepared statement object, assigning the results to a `ResultSet` object. This object now holds the results of the executed SQL query.

Step	Action	Description
5	**Retrieving Data**	Check for data in the `resultSet` using the `next()` method. If data exists, variables are created and assigned values from the first row of the table. The `next()` method positions us at the **1st row in the table** initially, and it's used again to move to subsequent rows.
6	**Assigning Values**	Assign values to variables by referencing **column names**, such as `id`, `name`, and `salary`. Values are taken from the **FIRST row** in the table, with the process repeated for additional rows as needed using the `next()` method.

The output will look like this:

```
ID: 1, Name: John Doe, Salary: 60000.0
```

If we want to **display all the elements on the screen**, we need to change the `if` condition to a `while` loop so that we process each element.

It will look like this:

```java
package codefinity;

import java.sql.*;

public class HibernateExample {
    static String jdbcUrl = "jdbc:mysql://localhost:3306/testDatabase";
    static String username = "db_username";
    static String password = "db_password";
    public static void main(String[] args) {

        try {
            Connection connection = DriverManager.getConnection(jdbcUrl
            PreparedStatement statement = connection.prepareStatement("
            ResultSet resultSet = statement.executeQuery();
            while (resultSet.next()) {
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                double salary = resultSet.getDouble("salary");
                System.out.println("ID: " + id + ", Name: " + name + ",
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Output:

```
ID: 1, Name: John Doe, Salary: 60000.0
ID: 2, Name: Jane Smith, Salary: 65000.0
ID: 3, Name: Jim Brown, Salary: 55000.0
ID: 4, Name: Jake Blues, Salary: 70000.0
```

Conclusion

For every interaction with a database using JDBC, **the same algorithm is used**, as shown in the example above. We won't use this exact algorithm frequently because **we will use Hibernate** to interact with the database, but it's essential for everyone to be aware of this method of database interaction **provided by the JDK**.

codefinity

📖 DAO

When we talk about programming with databases, the design pattern **DAO** is often mentioned. You may not be familiar with the term "*design pattern*", so let me briefly explain it.

Design Patterns

It's worth noting that design patterns are a **very important part of programming**, so I recommend studying them in more detail, as you will need them in your career!

> **Study more**
>
> **Design patterns** in programming are **reusable solutions** to common problems encountered in software design. They represent **best practices** that have been developed and refined by developers over many years.

The primary **goals** of using design patterns are:

1. **Code Reusability:** Instead of inventing a solution from scratch for every problem, developers can utilize proven and reliable patterns;
2. **Simplifying the Design Process:** Patterns offer ready-made templates for solving complex design challenges;
3. **Enhancing Code Readability and Maintainability:** Patterns make the code structure clearer and more understandable, making it easier to support and evolve.

Main types of design patterns

1. **Creational Patterns:** These patterns are related to the process of object creation, providing flexibility and reusability of existing code;

2. **Structural Patterns:** They assist in organizing classes and objects into larger structures;

3. **Behavioral Patterns:** Behavioral patterns manage complex control flows between objects.

It's also worth noting that design patterns are **learned through practice**. It will be challenging to memorize them purely in theory, so **every time we use a particular design pattern**, I will provide more information about it, and you will remember its usage through practical experience.

Significance in software development

The use of design patterns contributes to the creation of cleaner, more understandable, and maintainable code. They are **fundamental tools** in the arsenal of an experienced developer and play a key role in software architecture.

Data Access Object

The DAO pattern tells us that we should **abstract and restrict database access** from the code, separating the application's business logic and data access.

Advantages of using the DAO pattern:

- **Allows separating business logic** and data operations into different components, making it easier to develop and maintain the program;
- **Changes** in the database structure **do not affect** other parts of the program since data access is done through DAO interfaces, and vice versa;
- **DAO can be easily tested** using mock objects or simulation instead of a real database.

Understanding the DAO Pattern

Using the DAO pattern is akin to adopting a **three-tier architecture** for an application. Let's break down the roles of each tier:

First Tier: Data Access

- **Responsibility:** Interacts directly with the database;
- **Function:** Retrieves data and passes it to the service layer.

Second Tier: Service Layer

- **Responsibility:** Processes the retrieved data;
- **Operations:** For instance, it might involve decrypting an encrypted user password for usage in other service classes.

> **Note**
>
> All sensitive information, such as user passwords, is stored in the database in an encrypted format. We'll delve deeper into this when we explore **Spring Security**.

Third Tier: Controller Layer

- **Responsibility:** Prepares data for user transmission with minimal processing;
- **Details:** More information on the controller layer will be provided during our study of the **Spring framework**.

From this structural overview, it's evident that:

- Direct database access is confined to the first tier;
- Business logic resides in the service layer, as recommended by the DAO pattern.

Main Components of the DAO Pattern

In the upcoming sections of this course, we'll be looking at the main components of the DAO pattern that we'll utilize in our application development:

From this structure, it's clear that direct database access is limited to the lowest and first tier, while business logic is isolated in the service layer. This is roughly what the DAO pattern suggests.

Let's also take a look at the main components of the DAO pattern that we'll use when writing applications in this course:

- **DAO interfaces** are **interfaces** that describe the operations that can be performed with a database. They **define the structure of methods** for creating, reading, updating, and deleting data;
- **DAO implementations** are **classes** that implement DAO interfaces. They contain **specific code** for interacting with the database. These classes can use JDBC or other technologies to interact with data;
- **Entities** are **classes that reflect the structure of database tables**. They contain fields that represent the data that the program works with;
- **DAO Factory** is a factory method or a factory class responsible for **creating instances of DAO implementations**. It helps isolate the program code from specific DAO implementations.

Example

Let's take a look at the **approximate project structure** where we're working with the `employees` table. We need to **implement components** such as: 1. An interface that will specify the methods to be implemented for working with the database; 2. A class that implements this interface; 3. An entity class that will represent an object instance from the database.

```java
package codefinity.DAO;

import codefinity.model.Employee;

import java.util.List;

public interface EmployeeRepository {
    List<Employee> getAllEmployees();

    Employee getEmployeeById(Long id);

    //other CRUD operations
}
```

Here, at the top, you can see an **interface that defines the methods** to be used for interacting with the database.

```java
package codefinity.DAO.impl;

import codefinity.DAO.EmployeeRepository;
import codefinity.model.Employee;

import java.util.List;

public class EmployeeRepositoryImpl implements EmployeeRepository {
    @Override
    public List<Employee> getAllEmployees() {
        //implementation of this method
    }

    @Override
    public Employee getEmployeeById(Long id) {
        //implementation of this method
    }

    // other CRUD operations implementation
}
```

Above, you can see a **class that implements the interface** and provides the methods for interacting with the database.

```java
package codefinity.model;

import java.util.Date;

@Table(name = "employees")
public class Employee {
    private int id;

    private String name;

    private String position;

    private double salary;

    private Date hireDate;

    // constructors, getters and setters
}
```

Here, you can see an **entity class** whose attributes correspond to the columns of the `employees` table.

Summary

In this course, we will strive to use the **DAO design pattern** for developing our applications. Moving forward, we will explore **more advanced techniques** to make working with Java code and databases more convenient!

codefinity

📖 **Boilerplate Code Auto-Generation**

Space Saving in Java Code

Have you ever noticed how many **boilerplate things** are there in every Java code? For example, when you write a **model**, you need to declare a **no-argument** constructor, a **constructor with arguments**, **getters**, **setters**, override the `toString()` and `hashCode()` **methods**, and much more. Typically, this takes up **70+ lines of code**, making it difficult to spot additional logic in the class if it's present.

Conclusion: getters, setters, and additional boilerplate constructions **can clutter your code significantly**. However, there is an extension that effectively addresses this issue - **Project Lombok**.

Lombok

Study more

Project Lombok is a Java library that **automatically injects boilerplate code** into your code during compilation **using annotations**. This helps reduce boilerplate code, making the code more readable and easier to maintain.

To get started with Project Lombok, we need to **import it** into the `pom.xml` (if you're using Maven):

```
<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.30</version>
    <scope>provided</scope>
</dependency>
```

Great, now we can start improving our code.

Let's take a look at **some of the key annotations:**

Getter and Setter

- `@Getter` and `@Setter`: Generate standard getters and setters for a field. It can be applied at the field or class level.

Constructors

- `@NoArgsConstructor`: Generates a no-argument constructor;
- `@RequiredArgsConstructor`: Generates a constructor with one parameter for each field that requires special processing (such as fields annotated with `@NonNull` or final fields);
- `@AllArgsConstructor`: Generates a constructor with one parameter for each field of the class.

Let's take a look at some **code examples** of how to apply these annotations.

These annotations should be placed **above a class**:

```java
package com.example;

import java.util.Date;

public class Employee {

    private int id;

    private String name;

    private String position;

    private double salary;

    private Date hireDate;

    public Employee() {
    }

    public Employee(int id, String name, String position, double
salary, Date hireDate) {
        this.id = id;
        this.name = name;
        this.position = position;
        this.salary = salary;
        this.hireDate = hireDate;
    }

    public int getId() {
        return id;
    }
```

```
    public Employee(String name, String position, double salary) {
        this.name = name;
        this.position = position;
        this.salary = salary;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

    public String getPosition() {
        return position;
    }

    public void setPosition(String position) {
        this.position = position;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }
```

```
    public Date getHireDate() {
        return hireDate;
    }

    public void setHireDate(Date hireDate) {
        this.hireDate = hireDate;
    }
}
```

As you can see, this code **looks quite lengthy**, and it can be challenging to read anything in it, even though it's structured.

Let's improve it with the help of `Lombok` annotations:

```java
package com.example;

import lombok.*;

import java.util.Date;

@Getter
@Setter
@AllArgsConstructor
@RequiredArgsConstructor
public class Employee {

    private int id;

    @NonNull
    private String name;

    @NonNull
    private String position;

    @NonNull
    private double salary;

    private Date hireDate;
}
```

So, from 73 lines, we've reduced it to 25. Now, we can clearly see what **fields** are in this class, that there are **getters**, **setters**, a **constructor** that takes all arguments, and a constructor that takes 3 arguments: `name`, `position`, and `salary`. In this way, we have **significantly reduced the code** while retaining the same functionality.

Additional Annotations:

Project Lombok also provides additional annotations, such as those for overriding `toString()`, `equals()`, and `hashCode()`.

Let's take a look at some additional annotations that can be useful:

Here are some additional annotations in Markdown format:

Annotation	Description
`@ToString`	Generates a `toString()` method for the class.
`@EqualsAndHashCode`	Generates `equals(Object other)` and `hashCode()` methods.
`@NonNull`	Marks a field as not allowing null values and automatically generates corresponding checks.
`@Value`	Marks the class as immutable, adding `@Getter`, and making all fields `final` and `private`.
`@Data`	Combines `@Getter`, `@Setter`, and `@RequiredArgsConstructor`.

Now, let's further improve our code by replacing `@Getter`, `@Setter`, and `@AllArgsConstructor` with the `@Data` annotation, which is **placed before the `class` keyword**. Additionally, let's add the `@ToString` and `@EqualsAndHashCode` annotations.

The result will be the following code:

```java
package com.example;

import lombok.*;

import java.util.Date;

@EqualsAndHashCode
@ToString
@RequiredArgsConstructor
public @Data class Employee {

    private int id;

    @NonNull
    private String name;

    @NonNull
    private String position;

    @NonNull
    private double salary;

    private Date hireDate;
}
```

> **Note**
>
> We haven't covered all the annotations from the **Project Lombok** library. This chapter covers the **main annotations** and functions of this library. In the future, if we need to use a specific annotation that I haven't described in this chapter, I will explain separately what that particular annotation means.

In summary, Lombok is a **powerful tool** for optimizing Java code, but its **usage requires some caution**. All changes are made during compilation, and you'll need to install the Lombok plugin in your integrated development environment (IDE) to see these changes reflected. Additionally, it's worth noting that **excessive use of Lombok** can make it **challenging to read** and debug code if **developers are not accustomed to this style**.

In our course, **we will use Lombok** because we will be working extensively with entities, which take the form of model classes.

codefinity

Section 1 Summary

Great job!

The first section of the **database introduction** is complete; congratulations!

In this section, you've gone through the basic material on Java's interaction with databases. You've learned that this is done using **JDBC** and gained an understanding of **how Java** theoretically **interacts with various components**, especially with databases.

You've also been introduced to the **DAO pattern**, which we will use throughout this course.

Furthermore, you've gained a general understanding of what **design patterns** are and how to work with them.

Additionally, you've learned how to significantly improve your code using the **Project Lombok** library, which allows you to easily **generate boilerplate code** using annotations.

Next, you will get acquainted with the **Hibernate** framework, learn to work with it, and finally practice with assignments. There are many interesting and important topics ahead of you!

There won't be a quiz in this chapter, so you can move on :)