Simple Banking Java Maven Project

Simple Banking Application Documentation

---

1. Project Overview

The **Simple Banking Application** is a Java-based system that allows users to:

- Create a bank account with an initial deposit
- Deposit funds into the account
- Withdraw funds from the account
- Check account balance

How to Run:

Step 1: Clone the Repository

```
git clone https://github.com/AkhtargitHub/Simple-Banking-Java-Maven.git
```

Step 2: Navigate to the Project Folder

```
cd simple-banking
```

Step 3: Build and Run Tests

```
mvn clean verify
```

Step 4: Run the Application Manually

```
mvn compile exec:java -Dexec.mainClass="com.bank.Main"
```

CI/CD with GitHub Actions:

This project includes **GitHub Actions** to automatically run tests on each push or pull request.

---

2. Clean Code Practices Used

This project follows **clean code principles**, includes **JUnit 5 unit tests**, and is configured with **Maven** for dependency management. Additionally, **GitHub Actions** automate testing upon every push or pull request.

Key Clean Code Practices:

1. **Meaningful Variable & Method Names** - Clear naming conventions.
2. **Proper Exception Handling** - Prevents invalid transactions.

3. **Single Responsibility Principle** - Separate classes for logic and tests.
4. **Code Readability & Formatting** - Consistent indentation and comments.
5. **Test Coverage** - Comprehensive JUnit 5 tests for various scenarios.

Example 1: Meaningful Variable & Method Names

In `BankAccount.java`, the class and methods have clear and meaningful names that describe their purpose:

```
public class BankAccount {

    private String accountHolder;

    private BigDecimal balance;

    public BankAccount(String accountHolder, BigDecimal initialDeposit) {

        if (initialDeposit.compareTo(BigDecimal.ZERO) < 0) {

            throw new IllegalArgumentException("Initial deposit cannot be negative");

        }

        this.accountHolder = accountHolder;

        this.balance = initialDeposit;

    }

}
```

Why this is Clean Code:

- The class `BankAccount` has an **intuitive name**.
- `accountHolder` and `balance` **clearly describe their purpose**.
- Method names (`deposit()`, `withdraw()`, `getBalance()`) **indicate their functionality**.

Example 2: Proper Exception Handling & Input Validation

Proper validation ensures **no invalid transactions occur**:

```
public void withdraw(BigDecimal amount) {

    if (amount.compareTo(BigDecimal.ZERO) <= 0) {

        throw new IllegalArgumentException("Withdrawal amount must be positive");
    }

    if (amount.compareTo(balance) > 0) {

        throw new IllegalArgumentException("Insufficient funds");

    }

    balance = balance.subtract(amount);

}
```

Why this is Clean Code:

- **Prevents withdrawals** of negative amounts.
- Ensures users **cannot overdraft** their account.
- Uses **meaningful error messages** for better debugging.

Example 3: Separation of Concerns (Single Responsibility Principle)

- `BankAccount.java` **handles only business logic**.
- `BankAccountTest.java` **handles only testing**.

Each class serves a **single, well-defined purpose**, making the code **easier to maintain and extend**.

---

3. Explanation of Tests

The `BankAccountTest.java` class tests different banking scenarios:

Test Name	**Description**	**Expected Output**
`testInitialDeposit()`	Creates an account with $100.00	Balance = $100.00
`testDeposit()`	Deposits $50 into an account	Balance = $150.00
`testWithdraw()`	Withdraws $40 from an account	Balance = $60.00
`testWithdrawMoreThanBalance()`	Tries withdrawing $150 with only $100 in account	Exception: "Insufficient funds"

Example Test Code:

```
@Test

void testWithdrawMoreThanBalance() {

    BankAccount account = new BankAccount("Peter Pan", new BigDecimal("200.00"));

    Exception exception = assertThrows(IllegalArgumentException.class, () -> {

        account.withdraw(new BigDecimal("250.00"));

    });

    assertEquals("Insufficient funds", exception.getMessage());

}
```

---

4. Dependencies & Their Sources

This project uses **Maven** to manage dependencies. The required dependencies are in `pom.xml`:

```
<dependencies>

    <dependency>

        <groupId>org.junit.jupiter</groupId>

        <artifactId>junit-jupiter-api</artifactId>

        <version>5.7.0</version>

        <scope>test</scope>

    </dependency>

    <dependency>

        <groupId>org.junit.jupiter</groupId>

        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.7.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Dependency	Purpose	Source
JUnit 5 API	Unit Testing	Maven Central Repository
JUnit 5 Engine	Test Execution	Maven Central Repository

These dependencies are downloaded automatically when running:

```
mvn clean install
```

---

5. Problems Encountered & Solutions

Issue 1: Tests Not Running in IntelliJ IDEA

Problem: Tests were not being detected when running `mvn test`. **Solution:** Ensured `maven-surefire-plugin` was added to `pom.xml`:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
</plugin>
```

Issue 2: GitHub Actions Build Failed

Problem: GitHub Actions workflow failed due to missing JDK version. **Solution:** Explicitly set up JDK 11 in `.github/workflows/maven-ci.yml`:

```
- name: Set up JDK 11
  uses: actions/setup-java@v2
```

```
with:

  java-version: '11'

  distribution: 'temurin'
```

Issue 3: Floating Point Precision Errors in BigDecimal

Problem: Using `double` for currency values led to precision errors. **Solution:** Used `BigDecimal` instead for accurate financial calculations.

Conclusion

This project demonstrates **clean coding principles**, structured testing, and automated CI/CD. By following best practices, it is **maintainable, reliable, and extensible** for future enhancements.