Woodland Academy

5.3 Java データアクセス

- JDBC の基本
- ステートメント
- トランザクション
- DAO と DTO

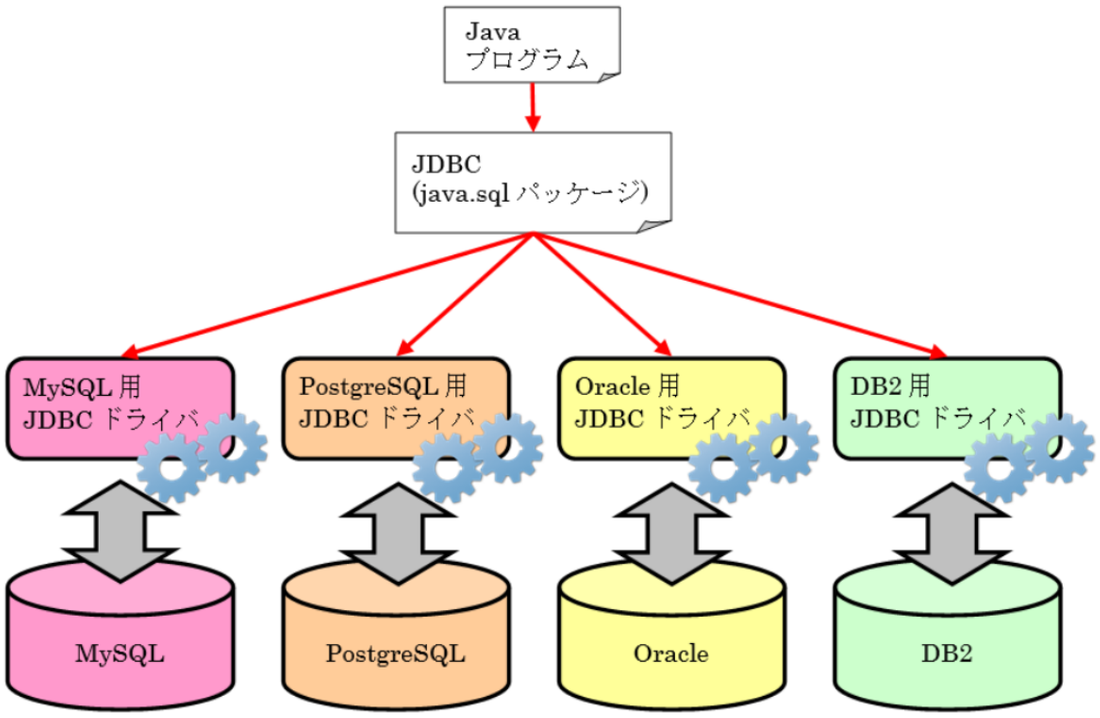
Shape Your Future

目次

JDBC とは

- **JDBC**（Java Database Connectivity）は、Java で異なる DBMS にアクセスするための**統一なインタフェース**です。各 DBMS のベンダーが、JDBC の具体的な実装（クラス）を提供してくれます。これらの実装のパッケージは**ドライバ**[Driver]と呼ばれます。

- 適切なドライバさえインストールされていれば、どの DBMS に接続しても同じ Java コードを書いてデータを処理することができます。



Woodland Academy

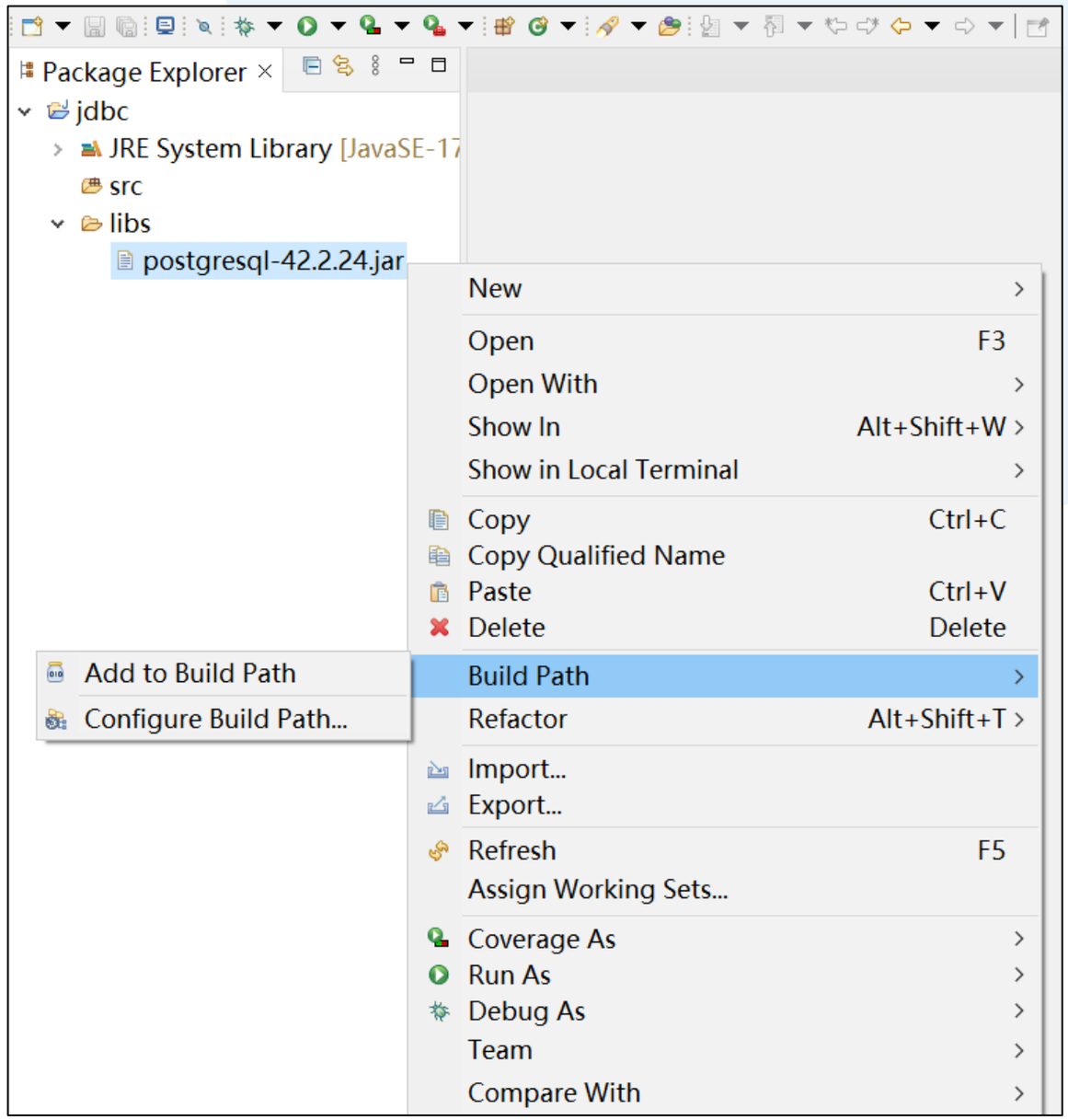
JDBC の使用手順

- JDBC の使用は 4 つのステップに分けられます：
  - ドライバの登録：Driver クラスを読み込む。
  - データベースの接続：データベースへの接続を確保。
  - データベース操作の実行：データベースに SQL 文を送る。
  - リソースの解放：データベースへの接続を閉じる。

Try
JDBCExample.java

ドライブの登録

- **ステップ 1**：対応する Driver クラスをコードに読み込むために、Driver の Jar パッケージをインポート：
  - postgresql-42.2.24.jar をプロジェクトの libs フォルダにコピー（フォルダがない場合は作成してください）。
  - Build Path を右クリック → Add to Build Path で、ドライバをプロジェクトに追加。



データベースとの接続

- **ステップ 2**： Java でデータベースに接続し、**Connection** オブジェクトを取得します：
  - データベースの**URL**、**ユーザー名**、**パスワード**など、データベースリンクに必要な情報を準備する。データベースの URL は：

```
jdbc:postgresql://localhost:[ポート番号]/[データベース名]
```

  - ここのポート番号は、登録時に設定した番号（デフォルトは 5432）。 例えば、前節の hello データベースの URL は以下のようになります:

```
String url = "jdbc:postgresql://localhost:5432/hello";
```

  - ユーザー名とパスワードは登録時に設定したものと同じです（デフォルトは postgres と 123456）。

- **Driver** オブジェクトを作成し、その **connect()** メソッドで接続を作成：

```
1 // 1. ドライバの登録と Driver オブジェクトの作成
2 Driver driver = new Driver();
3
4 // 2. データベースのリンクを取得する（方法 1: コードに直接設定を入力する）
5 // データベースのアドレスを設定する
6 String url = "jdbc:postgresql://localhost:5432/hello";
7 Properties info = new Properties();
8 info.setProperty("user", "postgres"); // ユーザー名の設定
9 info.setProperty("password", "123456"); // パスワードの設定
10
11 Connection con = driver.connect(url, info); // リンクの作成
```

設定情報の書き方

- JDBC でデータベースに接続する場合、設定の情報を文字列として直接コードに記述する場合があります：

```
String url = "jdbc:postgresql://localhost:5432/hello";

info.setProperty("password", "123456");
```

- このように設定情報を直接コードに書き込むと、2 つの問題があります：
  - コード内に重要な情報が露出し、情報セキュリティが危険にさらされている。
  - 設定の何かを変更したい場合、ソースコードを修正してコンパイルする必要があり、時間と労力がかかってしまうのです。

- これを解決する方法は、設定情報を外部ファイルに書き出し、そこに格納することです。

Woodland Academy

Properties の使用

- Java には **Properties** クラスがあり、「**.properties**」ファイルに設定情報を記述することで簡単にそれらを取得、保存、変更することができます。

- .properties ファイルでは、各行が 1 つの設定を表し、「**[設定名]=[設定値]**」の形式で記述されます：

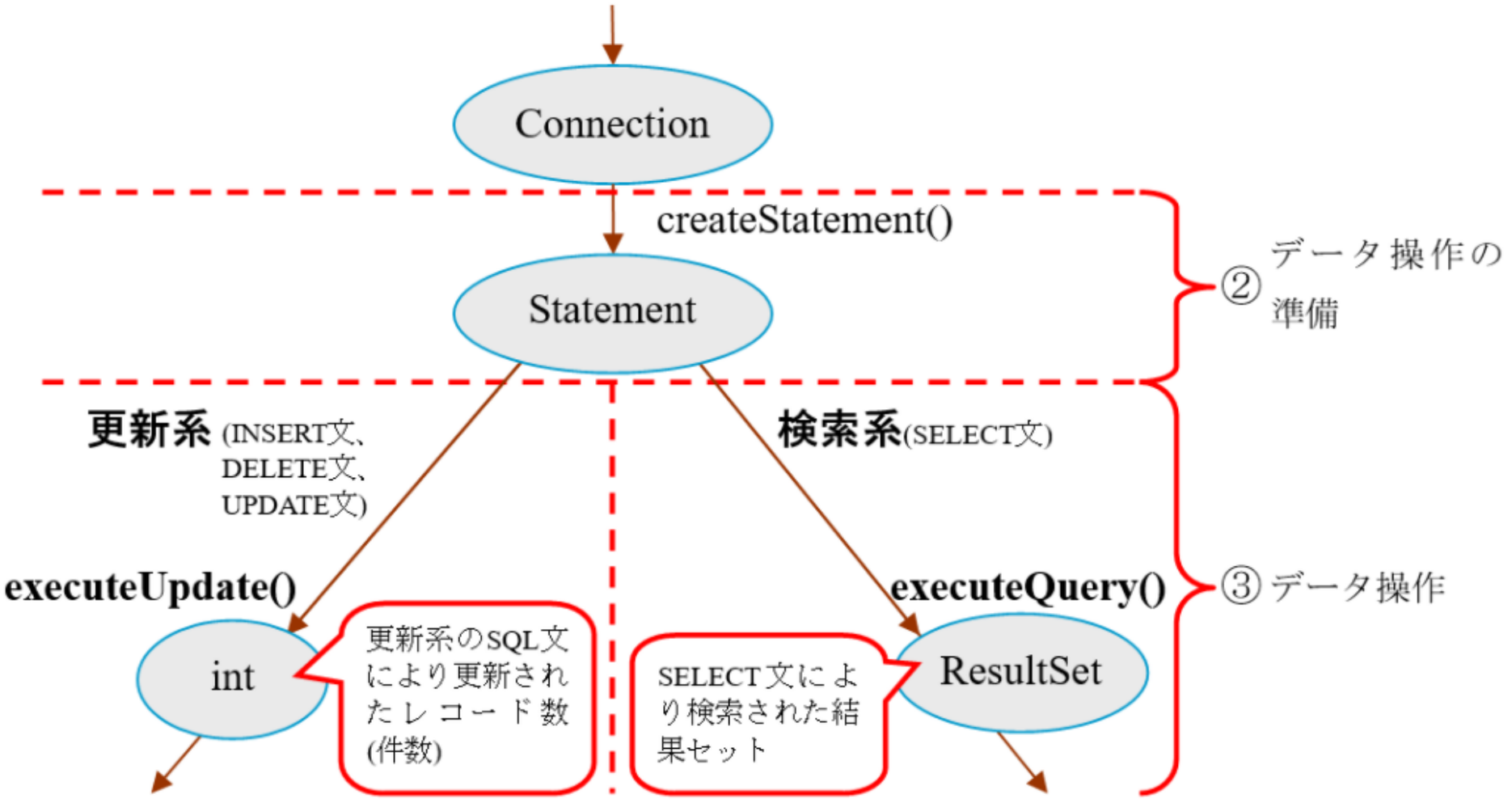
```
1 url=jdbc:postgresql://localhost:5432/hello
2 user=postgres
3 password=123456
```

- このような設定情報は、Properties クラスが提供するメソッドを通じて、直接読み込んで利用することができる:

```
properties.load(new FileReader("hello.properties"));
```

データベース操作の実行

- **ステップ 3**：データベースが接続されると、Connection クラスのメソッドを使って、追加、削除、変更、検索の操作を行うことができるようになります。



SQL クエリの実行

- SQL クエリを実行するには、まず Connection クラスの **createStatement()** メソッドで **Statement** オブジェクトを作成します：

```
Statement smt = con.createStatement();
```

- 次に、実行する SQL クエリを**文字列**に書きます：

```
String sql = "SELECT * FROM student";
```

次へ

Woodland Academy

前へ

- 最後に、この文字列を Statement オブジェクトの **executeQuery()** メソッドまたは **executeUpdate()** メソッドに渡して実行します：
  - executeQuery() メソッドは、検索（SELECT）の実行に使用されます。検索結果を保持する ResultSet オブジェクトを返します。
  - executeUpdate() メソッドは、更新（挿入、削除、変更）文を実行するために使用されます。 何行目が更新されたかを表す整数が返されます。

- 例えば、先ほどの SQL 文はクエリ文なので、executeQuery()メソッドを使います：

```
ResultSet rs = smt.executeQuery(sql);
```

Woodland Academy

データの検索

- **executeQuery()** は **ResultSet** オブジェクトを返します。

- ResultSet はクエリに関する多くの機能を持っていますが、今のところ、次の 2 つだけを覚えておけばよいでしょう：
  - **getXxx("column")**：現在の「閲覧している」レコードのフィールドを取得。 xxx はフィールドのデータ型、例えば getInt()、getString() などがある。
  - **next()**：次の行のレコードを「閲覧」し始める。

次へ

前へ

- 最初のときは、「0 行目」のレコードを「閲覧」します（つまり、まだ実際のデータを閲覧していない）。next() メソッドで次のレコードに移動し、そこから getXxx() メソッドで目的のフィールドを取得し、それを繰り返せばいい：

```
1 while (rs.next()) {
2     System.out.println("[id: " + rs.getInt("id")
3                        + ", name: " + rs.getString("name")
4                        + ", score: " + rs.getInt("score") + "]");
5 }
```

データの更新

- データの挿入（INSERT）、削除（DELETE）、変更（UPDATE）などの更新操作は **executeUpdate()** メソッドで行う。

- このメソッドは更新されたレコードの数を表す整数を返します（例：DELETE コマンドは、削除された行の数を返す）：

```
1 Statement smt = con.createStatement();
2
3 String sql = "DELETE FROM student WHERE name = 'Alice'";
4 int result = smt.executeUpdate(sql);
5
6 System.out.println(result); // => 1
```

リソースの解放

- **ステップ 4**：ファイルの読み書きなどの操作と同様に、作成した接続を閉じてシステムリソースを解放する必要があります。

- 接続を閉じるには、Statement および Connection クラスの **close()** メソッドを使用します：

```
smt.close();
con.close();
```

次へ

- リソースの釈放を確実に行うには、この前（　§ 2.5.3）学習した **try-with-resources** 文が使用できます（この方法では close() メソッドの使用は必要ありません）。

```
1 try (
2         Connection con = driver.connect(url, info);
3         Statement smt = con.createStatement();
4 ) {
5     // codes...
6 } catch (SQLException e) {
7     // Exception handling...
8 }
```

Woodland Academy

Q&A

目次

ステートメント

- 以前、Statement オブジェクトを使って SQL クエリを実行し、そのメソッドに渡された文字列が文の全体であることを説明しました。しかし、文字列操作でクエリ全体を直接作成すると、次のようなデメリットがあります：
  - 文字列の連結に複数の「+」を使用する必要がある、SQL クエリとJava 文が混在して、可読性が低く、ミスしやすい。
  - 文字列に直接パラメータを書き込むと、**SQL インジェクション**のリスクがあります。悪意のあるユーザーは、特殊なパラメータを渡すことで、データベースを破損させたり、データをさらしたりすることができる。
  - SQL 文の前処理ができず、実行効率に悪影響。

SQL インジェクション

- 実際にデータベースで操作する場合、ユーザーから提供されたデータをパラメータとして使用することが多いです。例えば、次のクエリは、ユーザーが提供したユーザー名を使用してパスワードをデータベースに照会します：

```
SELECT password FROM account
WHERE username = "何らかの入力データ";
```

次へ

- この場合、悪意のあるユーザーが SQL 言語を知っていて、サーバーのコードを推測すれば、特別なパラメータを書き込むことによって、元の文の意味を変えることができます：

```
1 SELECT password FROM account
2 WHERE username = '  '; UPDATE account SET password = 'HACKED  ';
```

- これは **SQL インジェクション**[SQL Injection]と呼ばれます。

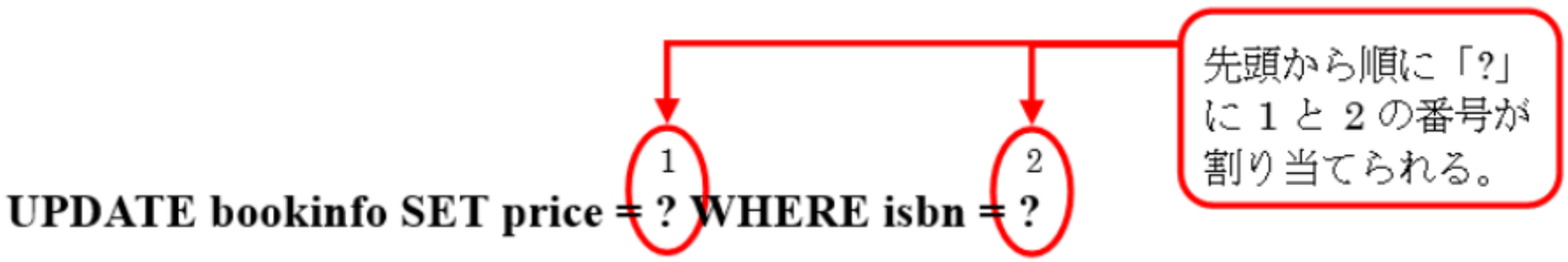
Try
SQLInjection.java

PreparedStatement

- これらの問題を回避するために、**PreparedStatement** クラスを使用して SQL クリエを実行することができます。

- PreparedStatement オブジェクトを作成するには、Connection オブジェクトの **prepareStatement()** メソッドを使用し、SQL の「テンプレート」を渡します：

```java
// SQL 文の「テンプレート」を用意する
String sql = "SELECT password FROM account WHERE username = ?";
// PreparedStatement オブジェクトを作る
PreparedStatement smt = con.prepareStatement(sql);
```

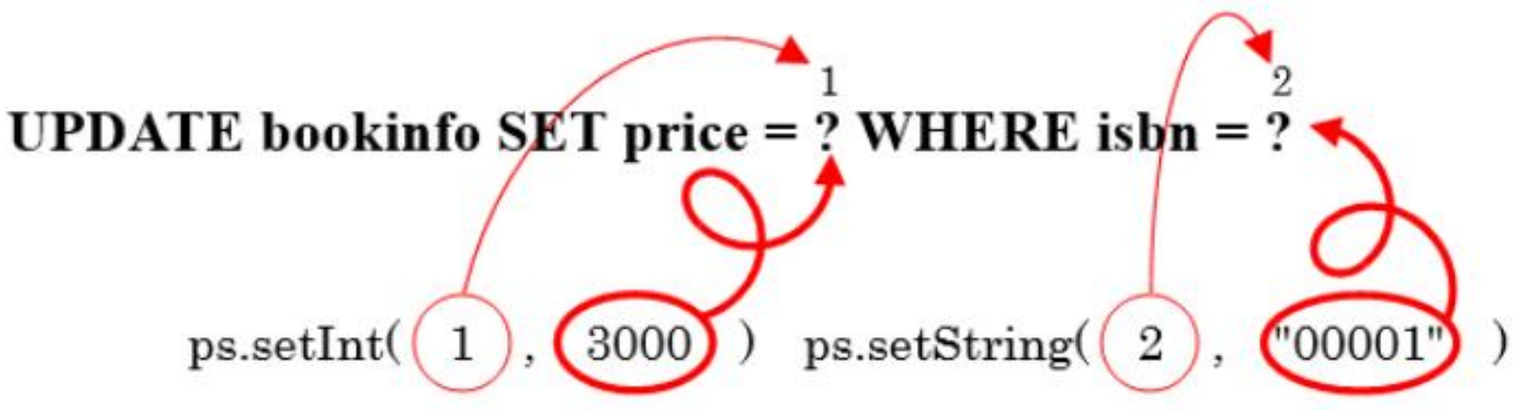


パラメータを設定

- 実際に使用するデータを取得（ユーザー入力など）した後、PreparedStatement の setXxx()メソッドを呼び出し、パラメータを設定。Xxx の部分は対応する変数の型。

- setXxx() メソッドは 2 つの引数があり、最初の引数は SQL の中のパラメータの番号を指定し、2 番目の引数はそのパラメータの値を指定します。

```
smt.setString(1, "Alice");
```



Note

パラメータ番号は 1 から始まる。

クエリを実行

- パラメータを設定できれば、executeQuery() メソッドや executeUpdate() メソッドを使うことができます。

```
ResultSet rs = smt.executeQuery();
```

Try

PreparedExample.java

- PreparedStatement は、SQL のパラメータを設定する前にデータの型をチェックするため、SQL インジェクションの問題を回避することができます。

Woodland Academy

Q&A

目次

データベーストランザクション

- **トランザクション**[Transaction]とは、連続した不可分な SQL クエリのグループです。

Example

銀行の引取データを操作する場合、複数の明細を持つ口座のお金を管理する必要があります。例えば、次のようなクエリを順番に実行することが必要：

1. 買い主の口座の金額を減らす。
2. 売主の口座に同じ金額を増やす。

これらの操作は不可分であるべきです。そうでなければ、もし操作が途中で失敗した場合、買い手が損をしたり、売り手が余計に得をすることになりかねません。

トランザクションの原則：ACID

- 一連のクエリをトランザクションに整理した後、以下の原則に従うこと：
  - **不可分性**[Atomicity]：トランザクションの各クエリは不可分であり、すべて完了するか、どれも完了しないのどっちであるべき。
  - **一貫性**[Consistency]：トランザクションの各クエリは、データベースが設定した条件（制約など）を満たす。
  - **独立性**[Isolation]：他の操作は、トランザクション全体の開始前または終了後の状態にのみアクセスでき、割り込むことはできない。
  - **永続性**[Durability]：トランザクションが完了した後、データに対する変更はデータベースに保存され、その結果は失われない。
- これらの原則は「**ACID**」と総称されます。

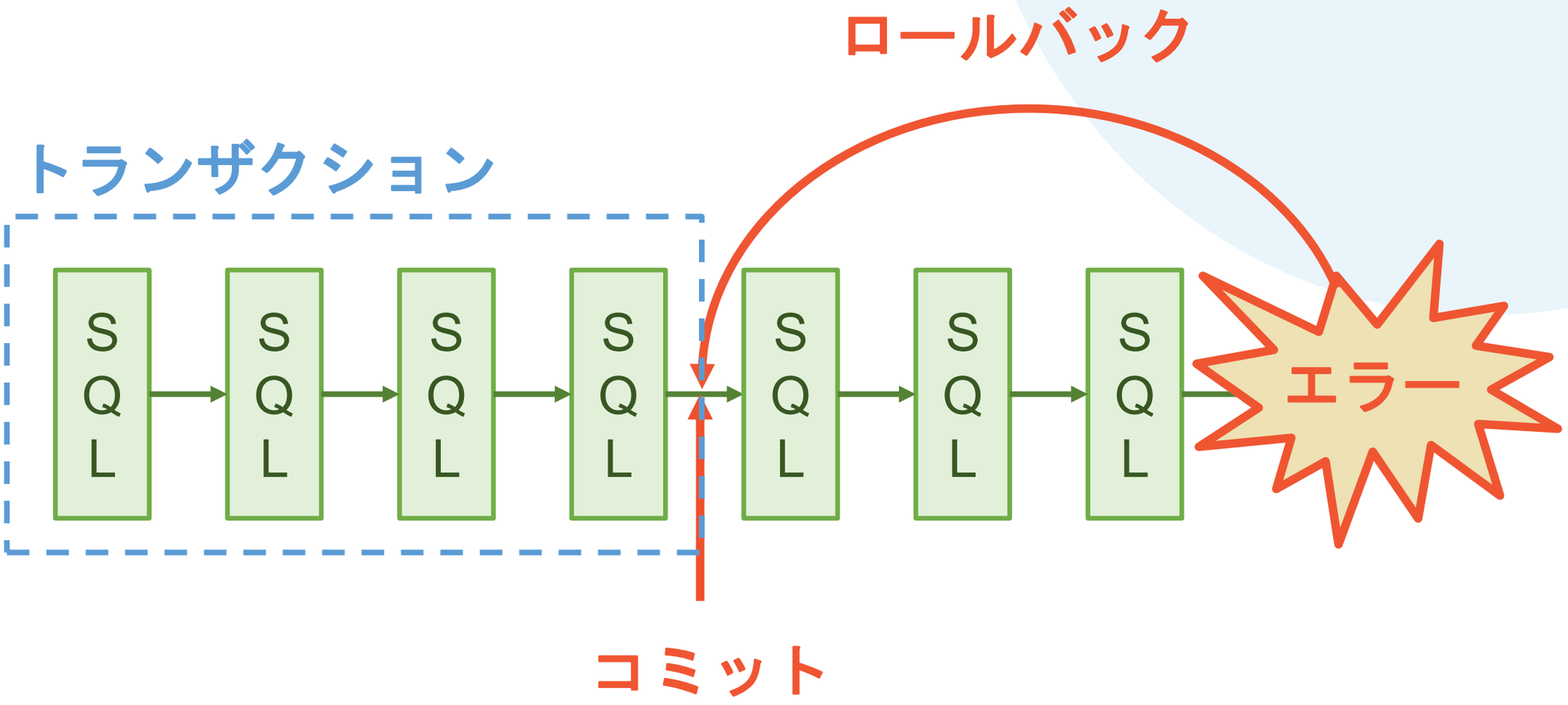
コミットとロールバック

- トランザクションの一般的な流れは、まずトランザクション内のすべての SQL クエリを順番に実行して、そして、**コミット**[Commit]操作を行います。

- つまり、2 回のコミット操作の間のすべてのクエリが、一つのトランザクションを構成します。

- トランザクションの実行中に何か問題が発生した場合、トランザクションを**ロールバック**[Rollback]することが必要です。ロールバックは、**コミットされていない**すべての SQL クエリ、すなわち最近のコミット以降に発生した全ての操作を取り消します。

次へ



JDBC のトランザクション

- デフォルトでは、JDBC の各操作は個別のトランザクションです。つまり、すべてのクエリは、実行後に**自動的にコミット**されます。

- 複数のクエリを一つのトランザクションにまとめるには、まず Connection オブジェクトの **setAutoCommit(false)** メソッドを呼び出してス自動コミット機能を無効化しなければなりません。

次へ

- その後、コミットせずにトランザクション内ですべての SQL クエリを実行することができます。これらのクリエの結果をコミットする必要があるときは、Connection の **commit()** メソッドを呼び出すればいいです。

- 操作に失敗したり例外が発生した場合に、Connection の **rollback()** メソッドを呼び出すと、トランザクション全体をロールバックすることができます。

Try

TransactionExample.java

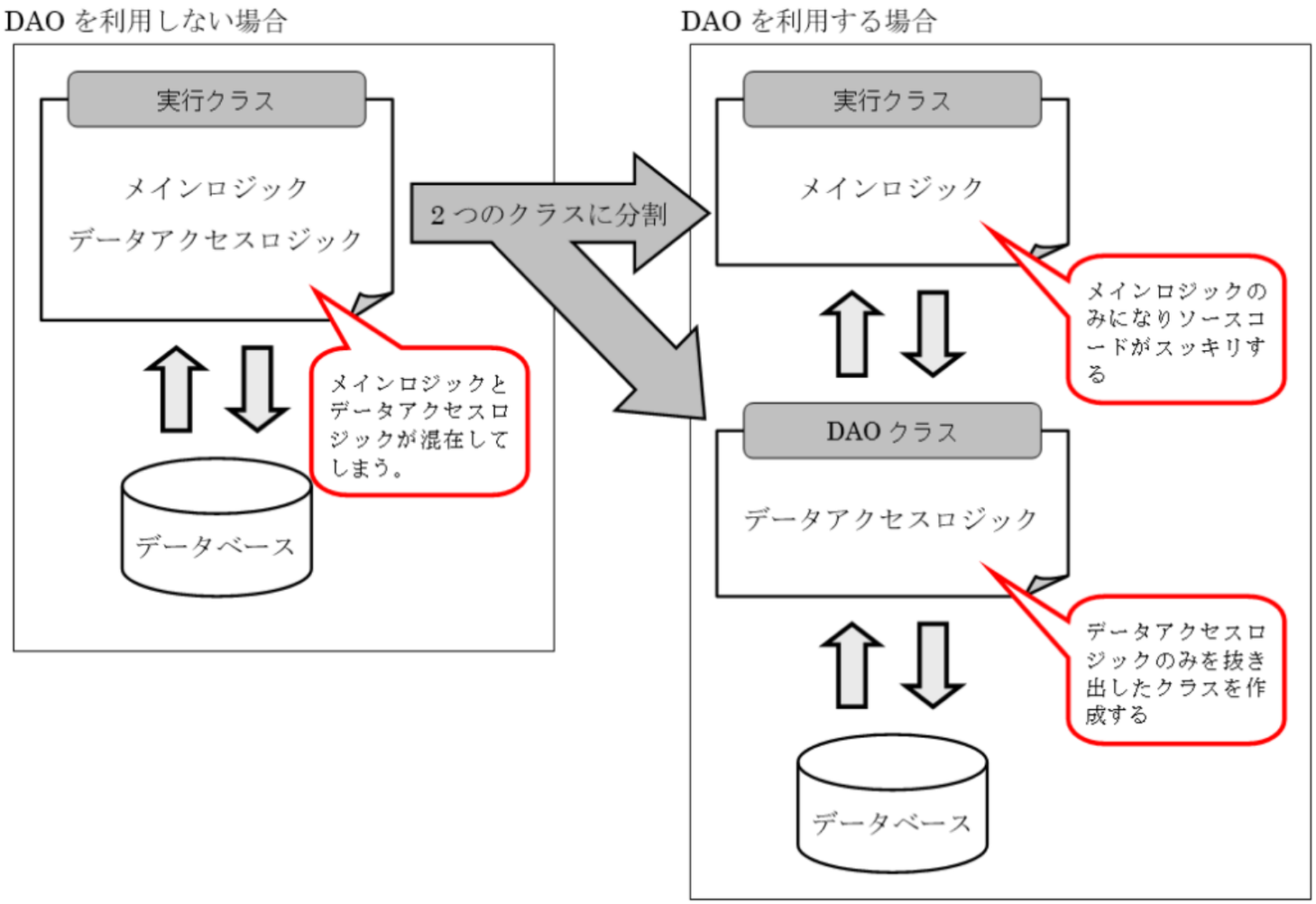
Woodland Academy

Q&A

目次

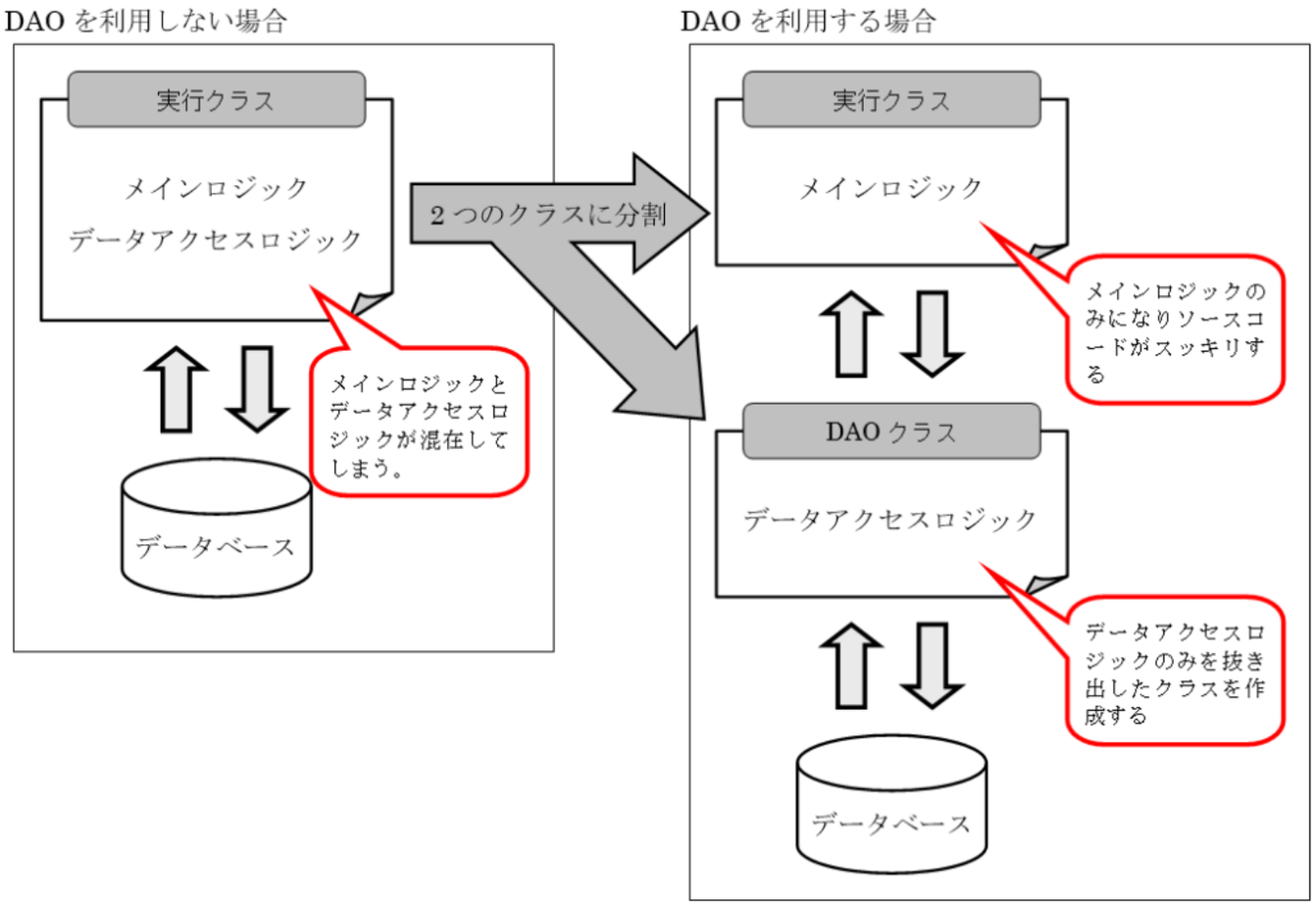
DAO パターン

- データベースを利用したアプリケーションを作成する際には、**DAO** と **DTO** の 2 つのプログラミングパターンがよく利用されます。



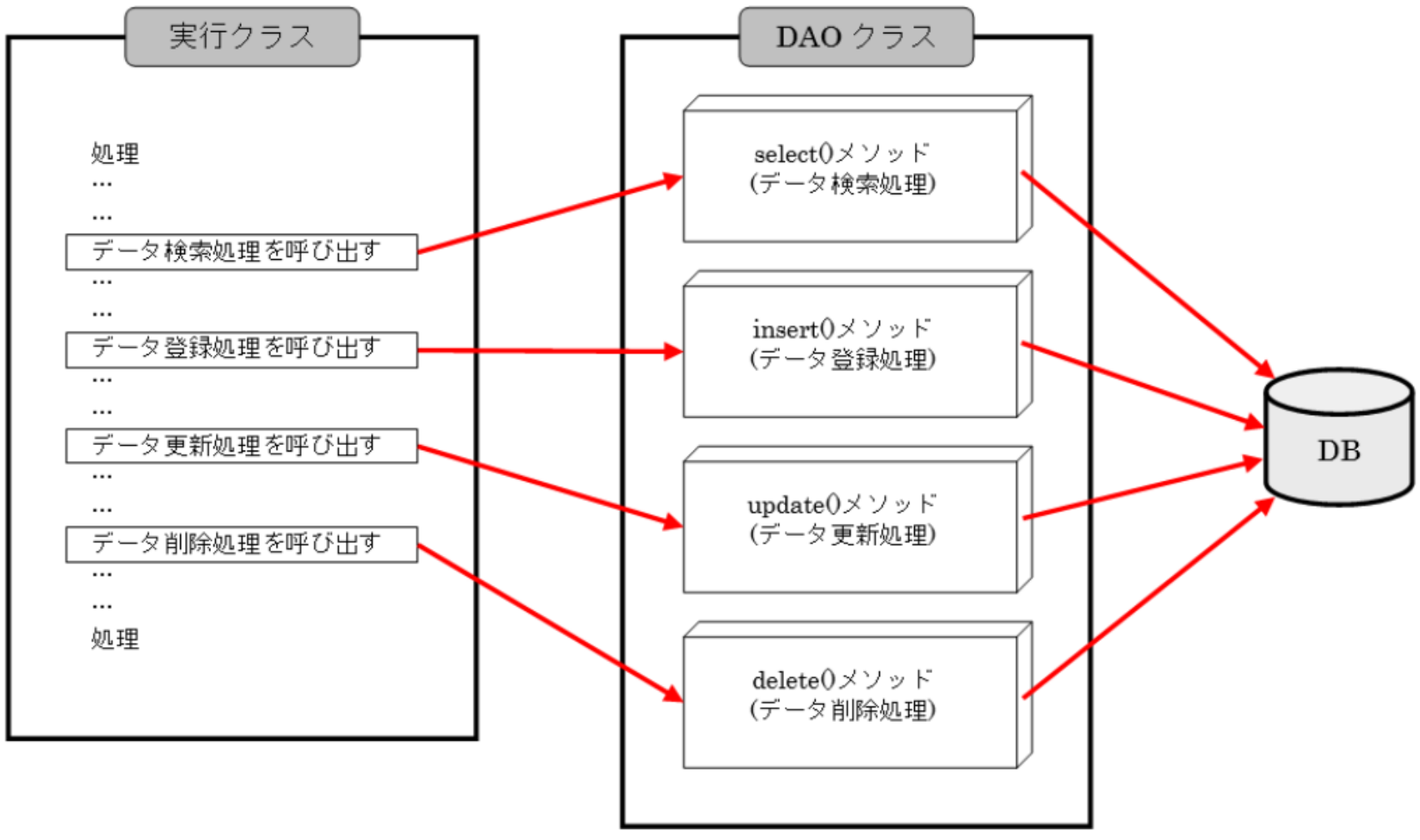
次へ

- **DAO**（Data Access Object）パターンとは、データベースにアクセスするための専用のクラスを作成し、データベースにアクセスするコードをプログラムのメインロジックから分離するパターンです。データベースへアクセスの「窓口」という意味で捉えることができます。



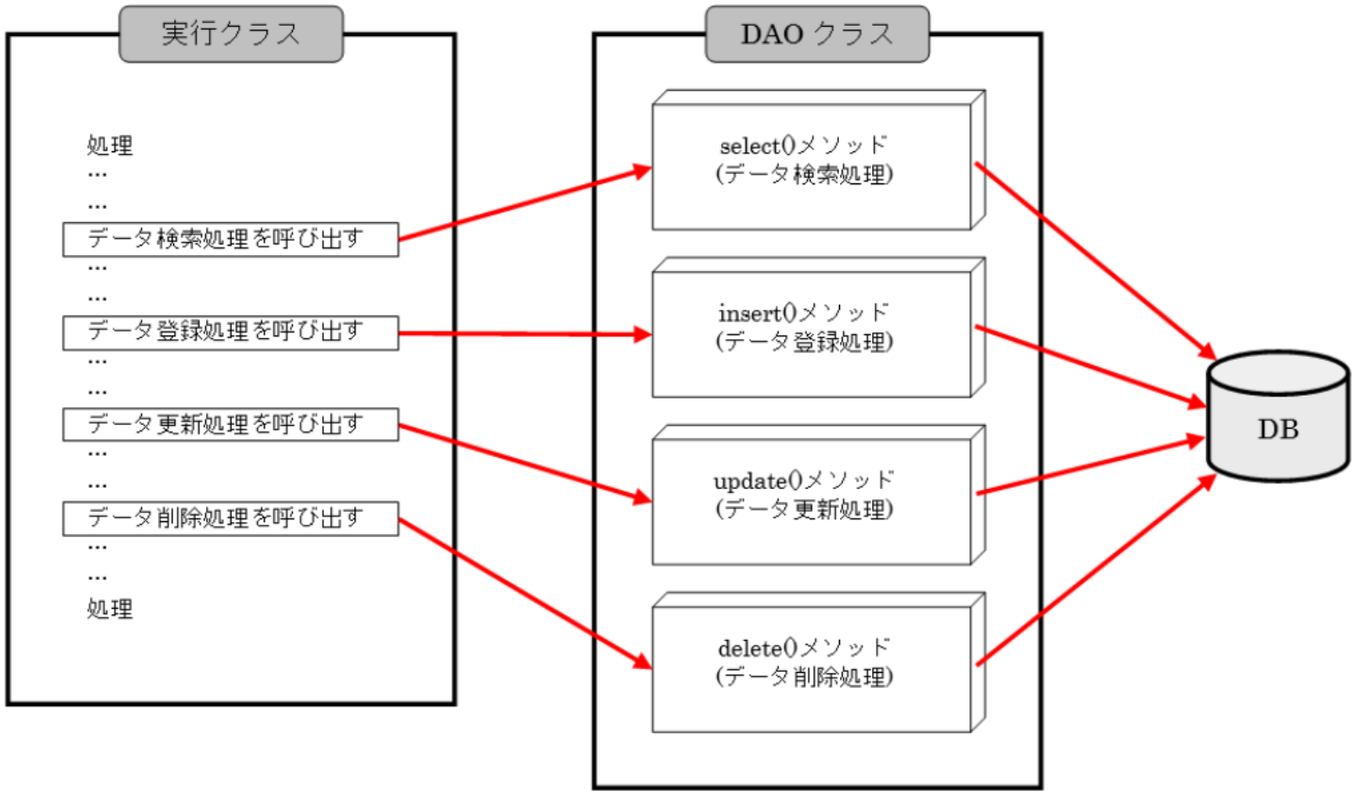
DAO モデルのメリット

- DAO パターンを使用したプログラムは、メインロジックとデータアクセス処理を明確に区別して設計されています。



次へ

- その結果、以下のような利点があります：
  1. 独立性・拡張性の向上。あるクラスを変更しても、他のクラスには影響しない（結合度を下がる）。
  2. 類似のデータベースアクセス機能を 1 つのクラスに統合し、可読性を向上させる。
  3. コードの簡素化：データベース処理の重複なコードを削減できる。



DTO モデル

- DAO パターンだけだと、データベースのフィールドごとにメソッドを定義する必要があるなど、メインロジックやデータアクセスに非効率な部分もあります。

- **DTO**（Data Transfer Object）パターンは、**JavaBeans** というコンセプトをベースにしている「データ転送専用のクラス」です。

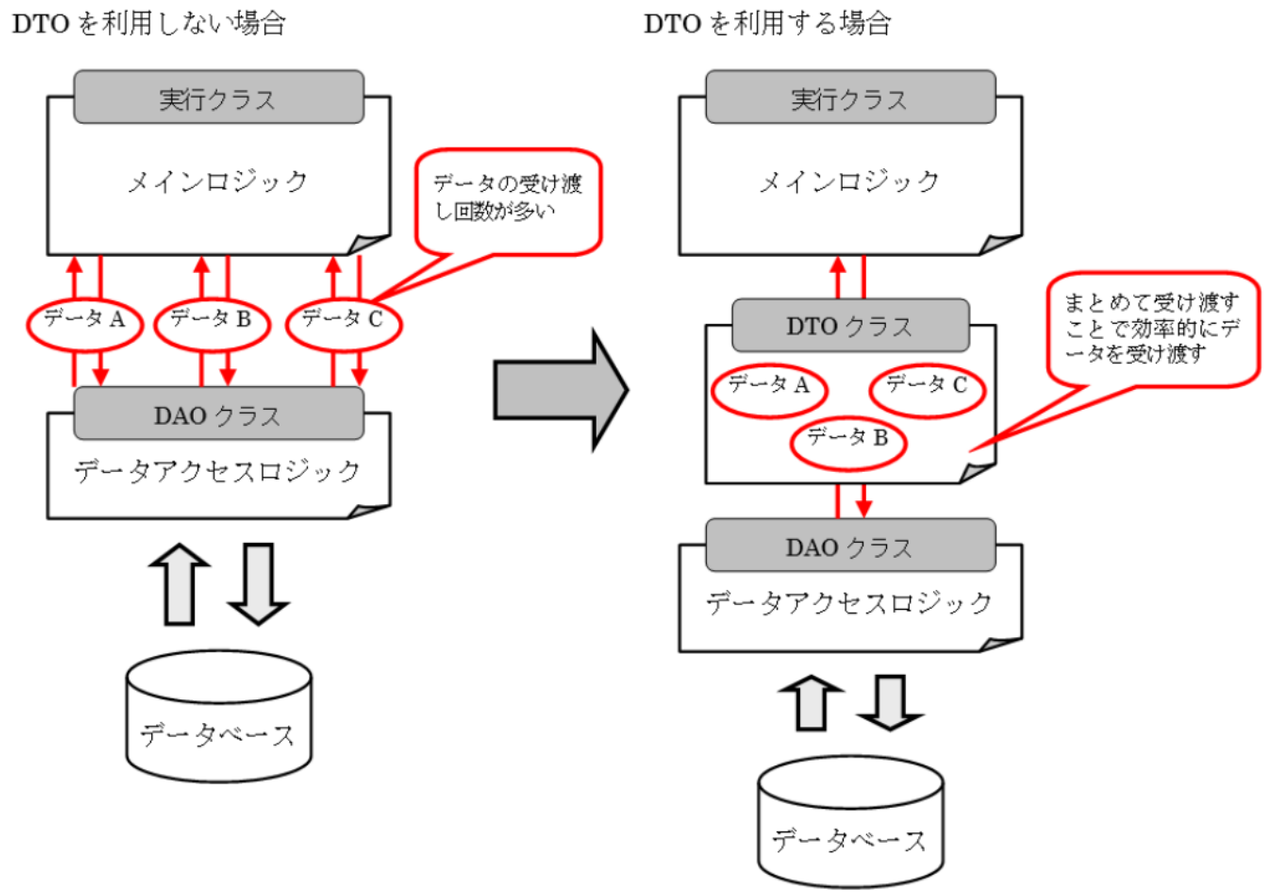
- 通常、データ全体（ユーザー、ブログなど）のクラスを定義し、そのフィールドごとに対応する変数と、変数のゲッターとセッターを定義します。データを転送する際に、データを個別に検索して転送するのではなく、**クラスごと**転送します。

DTO パターンのメリット

1. DTO パターンでは、データの転送は簡単で、一度に **1 つのオブジェクト**を転送すればよいのです。

2. オブジェクトをリストや配列などの**データ構造に配置**することで、さまざまなデータを管理することができます。

3. データベースへのアクセス回数が全体的に減り、**パフォーマンスが向上**する。



Try

student パッケージ

Woodland Academy

Q&A

まとめ

Sum Up

1.Java でデータベースの操作方法：JDBC。
① ドライバのインストールとインポート。
② 接続、ステートメントの作成、SQL クエリの実行の一連の流れ。

2.JDBC における SQL インジェクション の概念とその解決策：PreparedStatement。

3.トランザクション：ACID の原則、コミット、ロールバックの概念と JDBC での利用。

4.DAO と DTO パターンの基本概念。

Woodland Academy

Thank you!

From Seeds to Woodland — Shape Your Future.

Shape Your Future