

# JDBCプログラミング基礎

株式会社ナレッジエクス  
<http://www.knowledge-ex.jp/>

Version 0.9.003

# 本ドキュメントについて



- この作品は、クリエイティブ・コモンズの表示-改変禁止 2.1 日本ライセンスの下でライセンスされています。この使用許諾条件を見るには、<http://creativecommons.org/licenses/by-nd/2.1/jp/> をチェックするか、クリエイティブ・コモンズに郵便にてお問い合わせください。住所は: 559 Nathan Abbott Way, Stanford, California 94305, USA です。
- 本ドキュメントの最新版は、<http://www.knowledge-ex.jp/opendoc/jdbc.html> より入手することができます。

あなたは以下の条件に従う場合に限り、自由に



本作品を複製、頒布、展示、実演することができます。

あなたの従うべき条件は以下の通りです。



**表示.** あなたは原作者のクレジットを表示しなければなりません。



**改変禁止.** あなたはこの作品を改変、変形または加工してはなりません。

- 再利用や頒布にあたっては、この作品の使用許諾条件を他の人々に明らかにしなければなりません。
- 著作[権]者から許可を得ると、これらの条件は適用されません。
- Nothing in this license impairs or restricts the author's moral rights.

# Agenda

- JDBCの基本
- JDBC API詳細

# JDBCの基本

株式会社ナレッジエクス  
<http://www.knowledge-ex.jp/>

# JDBCの基本

- JDBCの基本
  - JDBCとは
  - JDBCの特徴
  - JDBCの構成
  - JDBCドライバ

# JDBC (Java Database Connectivity) とは

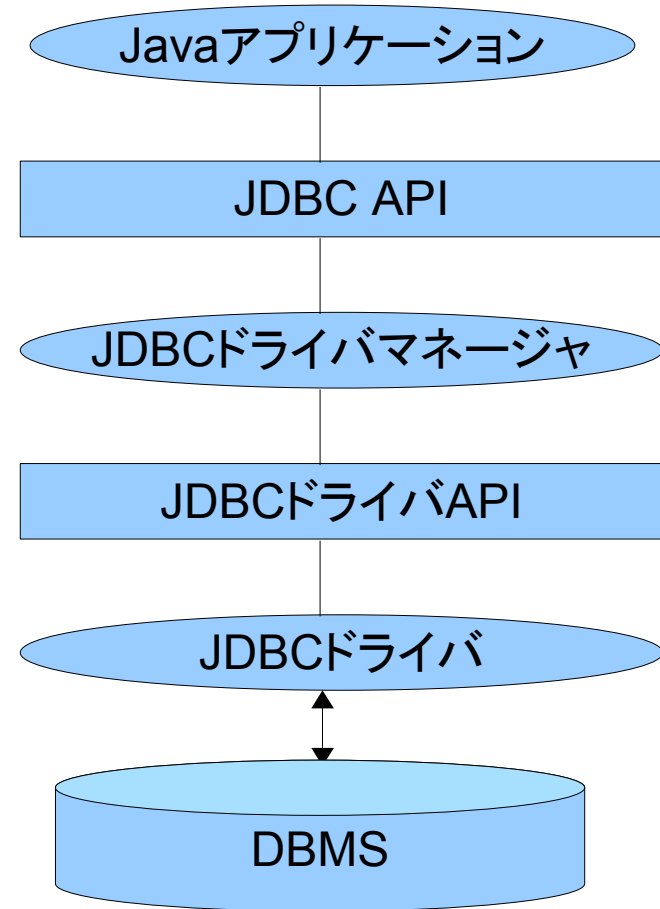
- テーブル形式のデータ(RDBMS)にアクセスするためのJava API
  - RDBMS = Relational Database Management System
- Java言語でDBMSにアクセスする標準的手段
  - 標準APIに含まれている (java.sqlパッケージ)
- どのDBMSに対しても同じAPIでアクセスできる
  - 同じ文法体系で多種のDBMSに対応可能

# JDBCの特徴

- DBMSに依存しないデータアクセスが可能
  - DBMSごとに操作クラス名などを変えなくて良い
- SQL文の発行が容易
  - 文字列で記述したSQL文をJavaのメソッドで発行できる
  - パラメータ付きのSQLも安全に発行できる
- SQLとJavaのデータ型をマッピングしてくれる
  - データ型に合わせた取得用メソッドが用意されている

# JDBCの構成

- JDBC API
- JDBCドライバマネージャ
- JDBCドライバAPI
- JDBCドライバ



# JDBC API

- アプリケーションで利用されるAPI
  - プログラマがJDBC操作に使用するAPI
  - JDBCのコーディング = JDBC APIの習得

# JDBCドライバマネージャ

- JDBCドライバの登録、維持などの管理を行う
  - 複数のJDBCドライバを管理することも可能
  - アプリケーションはドライバ管理の複雑な処理コードを記述する必要がなくなる

# JDBCドライバAPI

- JDBCドライバを操作するためのAPI
  - JDBCドライバを開発する場合には、これらのAPIが必要となるが、アプリケーションプログラミングでは不要

# JDBCドライバ

- DBMSに対する直接的な制御処理を行う
  - データベースサーバへの接続・解放
  - SQL文の送信
  - 実行結果の取得
  - などをJDBC APIの呼び出しに基づき行う

# JDBCドライバの入手と利用

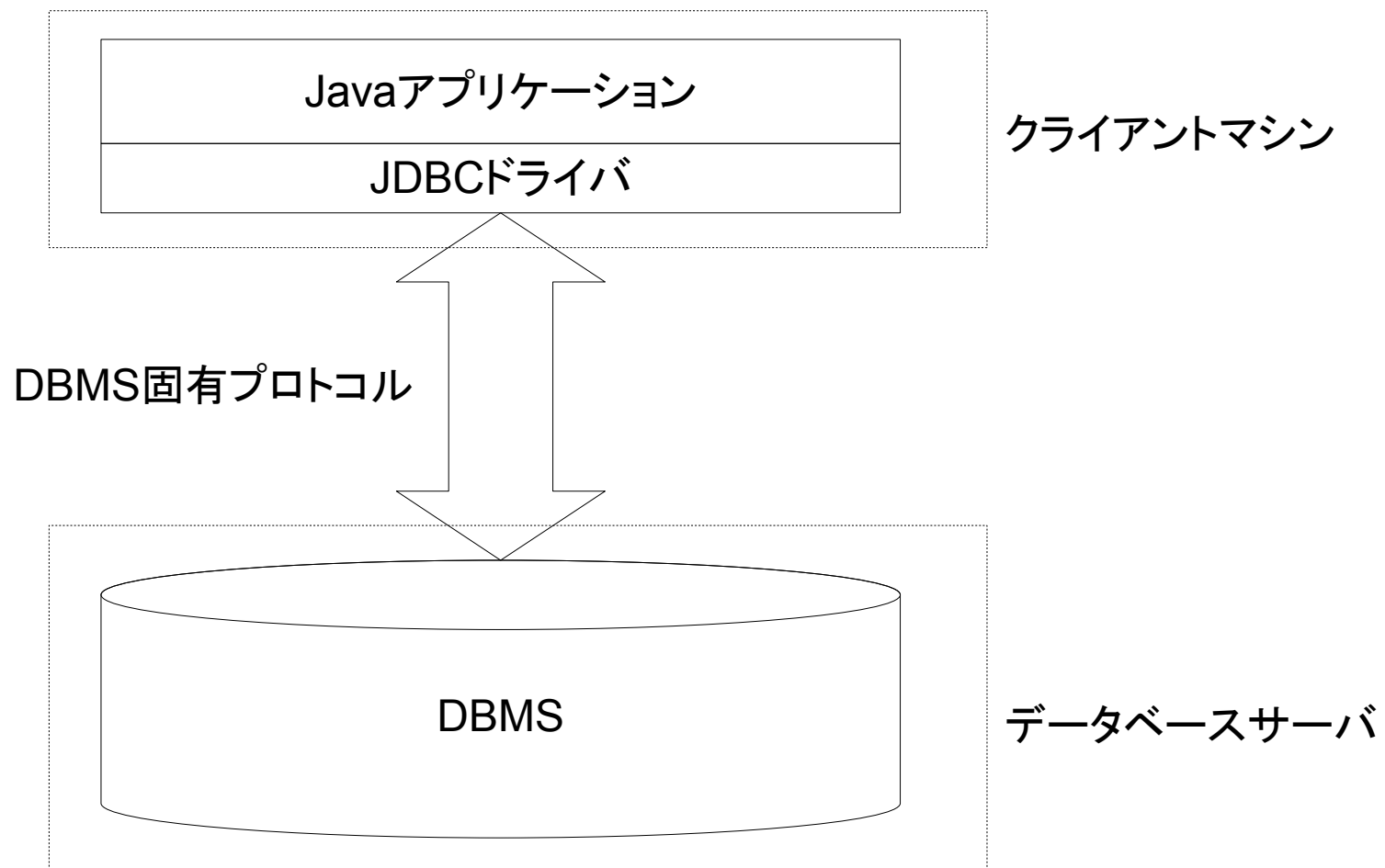
- JDBCドライバの入手方法
  - JDBCドライバは各DBMS毎に異なる
    - 基本的には各DBMSベンダーが提供するものを入手
  - JAR形式、ZIP形式で提供されることが多い
- JDBCドライバの利用
  - 実行時のCLASSPATHに入手したJDBCドライバのファイルパスを追加しておく
  - コンパイル時にはCLASSPATHへの追加は必須ではない

# JDBCの利用形態

- 2層モデル
  - 各クライアントからDBMSにアクセスするモデル
  - クライアント/サーバモデルで用いられる形態
- 3層モデル
  - 各クライアントが直接ではなく、サーバ層に配置されるアプリケーションサーバなどがDBMSにアクセスするモデル
  - Webアプリケーションモデルで用いられる形態

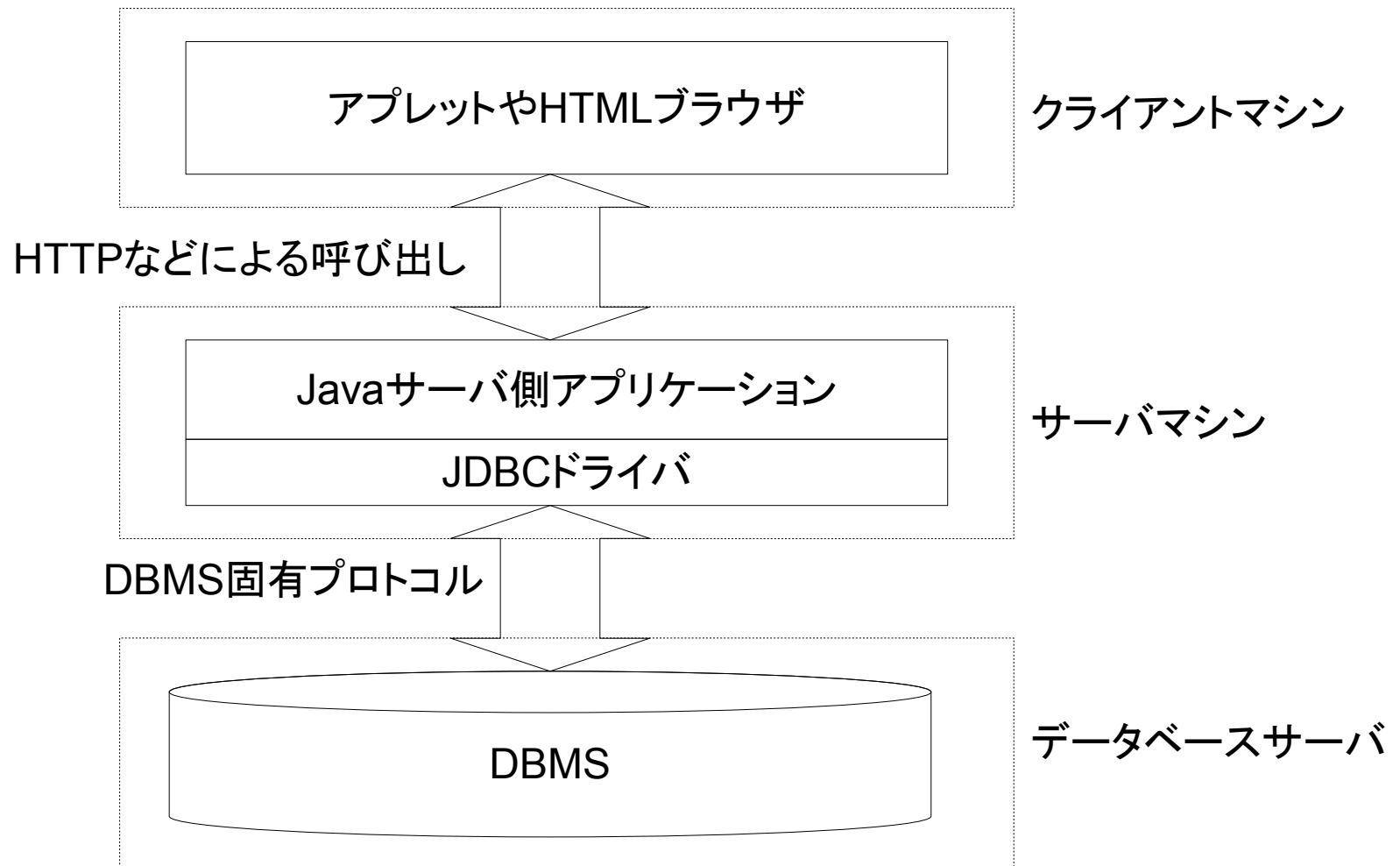
# 2層モデルでのJDBCの利用

- 2層モデルでは、各クライアントマシンにJDBCドライバを配置する必要がある



# 3層モデルでのJDBCの利用

- 3層モデルでは、JDBCドライバはクライアントマシンではなくサーバ層（アプリケーションサーバ）に配置

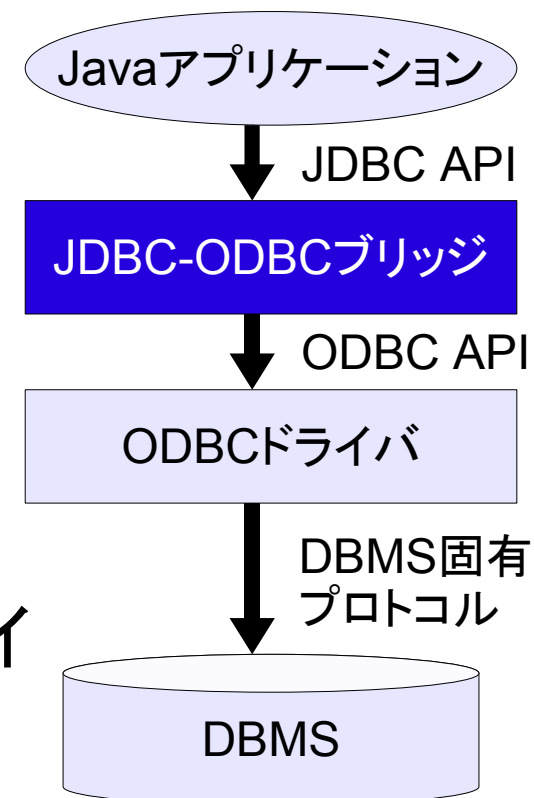


# ドライバの種類

- TYPE1／JDBC-ODBCブリッジ
- TYPE2／ネイティブブリッジ
- TYPE3／ネットドライバ
- TYPE4／ダイレクトドライバ

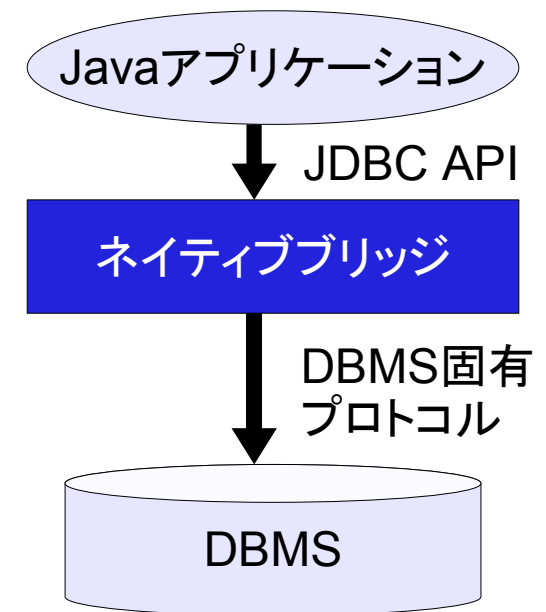
# TYPE1/JDBC-ODBCブリッジ

- JDBC呼び出しをODBC呼び出しに変換してアクセス(ブリッジ=橋渡し)するドライバ
  - JDBCのリリース当初はJDBCドライバが少なかったため、先行して市場に普及していたODBCを利用するために考えられたドライバ(過渡的な役割)
  - ドライバ自体にOSネイティブなコードを含むため、機種依存性がある
  - JDKに標準添付されている
  - 呼び出しの変換があるため、理論的に効率が期待できず、性能もODBCドライバに依存する



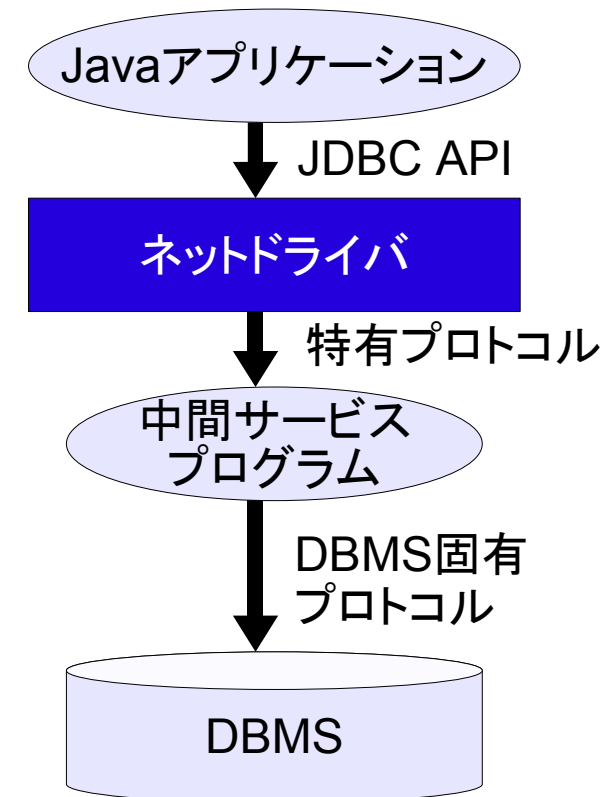
# TYPE2 / ネイティブブリッジ

- JDBC呼び出しを各DBMS固有のAPIに変換してアクセスするドライバ
  - 基本的な原理はJDBC-ODBCブリッジと同様
  - ドライバ自体にOS/DBMSネイティブなコードを含むため、機種依存性がある
  - 各DBMS固有のAPIで呼び出すため、Javaの黎明期には、高速アクセスが期待できることがメリットとなった



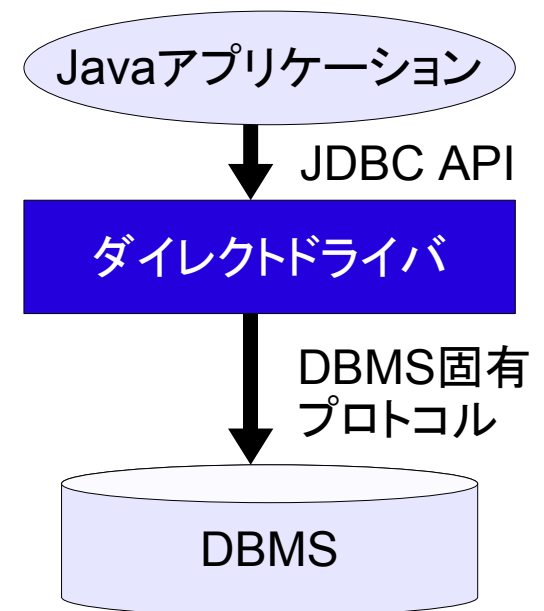
# TYPE3 / ネットドライバ

- ネットワーク上（主にサーバ層）に「中間サービスプログラム」を起動しておき、これを経由してアクセスするドライバ
  - 「中間サービスプログラム」とはクライアントからのJDBC API呼び出しをリモートで受けて、データベースアクセスを仲介するもの
  - これをサーバ層に配置することで、クライアント側のデータアクセス処理の負荷を軽減することを狙ったもの
  - あらかじめ中間サービスプログラムを起動しておく必要がある
  - 3層モデルの構造と同様なので、現在はほとんど使われない



# TYPE4:ダイレクトドライバ

- JDBC APIをJavaで直接DBMS固有プロトコルに変換してアクセスするドライバ
  - ドライバ自体が全てJavaで記述されているため機種依存性がない
  - 接続構造が簡素で、JDBCドライバ以外のドライバやサービスが不要
  - 性能はJava実行環境の性能に依存
  - 現在主流の形式



# JDBC API詳細

株式会社ナレッジエクス  
<http://www.knowledge-ex.jp/>

# JDBC API詳細

- 基本的な利用手順
  - JDBCドライバの登録
  - DBMSへの接続
  - SQLの発行
  - 結果の参照
  - 接続の解除
- 応用的な利用手順
  - プリコンパイル
  - トランザクションの扱い
  - メタデータクラス

# JDBC APIのパッケージとクラス群

- パッケージ
  - java.sqlパッケージ (Java標準API)
- クラス
  - java.sql.DriverManager
  - java.sql.SQLException
- インターフェース
  - java.sql.Connection
  - java.sql.Statement
  - java.sql.ResultSet
  - java.sql.PreparedStatement
  - java.sql.CallableStatement

# JDBCドライバの登録(1)

- JDBCドライバを利用するには、あらかじめDriverManagerクラスに登録する必要がある
  - 登録手順(1)
    - 実行対象のJavaVMにJDBCドライバのメインクラスをロードし、DriverManagerクラスにそのインスタンスを登録する
  - 登録手順(2)
    - 起動時のJVMオプションにシステムプロパティでJDBCドライバのメインクラス名を指定しておくこと、自動でそのクラスがロードされ、DriverManagerクラスにそのインスタンスが登録される

# JDBCドライバの登録(2)

- 登録手順(1)

- Class.forNameメソッドを使用しドライバのロードとインスタンス化を行う
  - 引数にドライバのクラス名を指定する
  - ドライバのメインクラス名はドライバ毎に異なる

コード例

```
Class.forName("com.mysql.jdbc.Driver");
```

# JDBCドライバの登録(3)

- 登録手順(2)

- JVM起動時のオプションでシステムプロパティ「jdbc.drivers」にドライバのメインクラス名を指定する
- システムプロパティは-Dオプションで指定  
例:java -Djdbc.drivers=com.mysql.jdbc.Driver

## コマンドライン例

```
java -Djdbc.drivers=com.mysql.jdbc.Driver 起動クラス名
```

# (参考) 主要DBMSのドライバクラス名

DBMS	バージョン	JDBCドライバクラス名
DB2 UDB	8.1～	com.ibm.db2.jcc.DB2Driver
MySQL	3.1～	com.mysql.jdbc.Driver
Oracle	8～	oracle.jdbc.driver.OracleDriver
PostgreSQL	7.2～	org.postgresql.Driver
SQLServer	～2000	com.microsoft.jdbc.sqlserver.SQLServerDriver
	2005～	com.microsoft.sqlserver.jdbc.SQLServerDriver
ODBCブリッジ		sun.jdbc.odbc.JdbcOdbcDriver

# JDBCドライバの登録コード例

## コード例

```
try {
    Class.forName("com.mysql.jdbc.Driver");
    ...
} catch (ClassNotFoundException ex) {
    ex.printStackTrace();
}
```

※ Class.forNameメソッドは、引数に指定されたクラスが見つからなかった場合に、例外ClassNotFoundExceptionを送出するため、例外処理が必要

# DBMSへの接続

- DriverManagerクラスのgetConnectionメソッドを実行
  - 引数 = ①「JDBC URL」 ②「ID」 ③「パスワード」
  - 戻り値 = java.sql.Connectionオブジェクト
- Connectionオブジェクト
  - DBMSへの接続を抽象化したオブジェクト
  - アプリケーションはこのオブジェクトを通じてDBMSに接続を行う

# JDBCのURL

- データベースの位置を特定するための文字列

標準的な構文

jdbc:subprotocol:subname

DBMSの識別子、アドレス、パラメータ  
など(JDBCドライバごとに異なる)

JDBCドライバの識別子

プロトコル名

# 主要DBMSのJDBC URL

DBMS	バージョン	JDBC URLの書式
DB2 UDB	8.1～	jdbc:db2://ホスト名:ポート番号/データベース名
MySQL	3.1～	jdbc:mysql://ホスト名:ポート番号/データベース名
Oracle	8～	jdbc:oracle:thin:@ホスト名:ポート番号:データベース名
PostgreSQL	7.2～	jdbc:postgresql://ホスト名:ポート番号/データベース名
SQLServer	～2000	jdbc:microsoft:sqlserver://ホスト名:ポート番号;DatabaseName=データベース名
	2005～	jdbc:sqlserver://ホスト名:ポート番号;DatabaseName=データベース名
ODBCブリッジ		jdbc:odbc:データソース名

※ ホスト名=サーバ名またはIPアドレス

※ デフォルトのポート番号を用いている場合、「:ポート番号」の指定は省略可能

# JDBCドライバの登録コード例

## コード例

```
try {
    String url = "jdbc:mysql://localhost/companydb";
    String id = "root";
    String pass = "passwd";
    Connection conn = DriverManager.getConnection(url, id, pass);
} catch (SQLException ex) {
    ex.printStackTrace();
}
```

※ DriverManager.getConnectionメソッドは、接続に失敗した場合に、例外java.sql.SQLExceptionを送出するため、例外処理が必要  
(java.sqlパッケージのメソッドは他のメソッドもほとんどのものがSQLExceptionを送出するため、同様の処理が必要)

# 演習問題(1)

次のような設定のデータベースがあります。

DBMSの種類	MySQL
データベース名	companydb
ホスト名	localhost
接続用ID	myuser
接続用パスワード	mypass

- (問1) Class.forNameメソッドを使ってデータベースに接続し、接続を確立 (Connectionオブジェクトを取得すること) してください。
- (問2) システムプロパティを使ってデータベースに接続し、接続を確立してください。

# SQLとは

- SQLとは
  - RDBMSにおいてデータの操作や定義等を行うための言語
  - ANSI / ISOで言語仕様が標準化されている
- JDBCとSQL
  - JDBC APIには文字列引数としてSQL文を指定することのできるメソッドがあり、SQL文を利用してDBMSにアクセスすることができる

# SQLを実行するためのAPI

- Statementインタフェース
  - プリコンパイルなどをしない単純なSQLに使用
- PreparedStatementインタフェース
  - プリコンパイルが必要なSQLに使用
- CallableStatementインタフェース
  - ストアドプロシージャを実行する場合に使用

# Statementオブジェクトの生成

- ConnectionオブジェクトのcreateStatementメソッドを実行
  - 引数=なし
  - 戻り値=java.sql.Statementオブジェクト

## コード例

```
try {  
    ...  
    Connection conn = DriverManager.getConnection(url, id, pass);  
    Statement stmt = conn.createStatement();  
    ...  
} catch (SQLException ex) {  
    ex.printStackTrace();  
}
```

※ Connection.createStatementメソッドは、接続に失敗した場合に、例外java.sql.SQLExceptionを送出するため、例外処理が必要 (java.sqlパッケージのメソッドは他のメソッドもほとんどのものがSQLExceptionを送出するため、同様の処理が必要)

# 2種類のSQL

- 参照系SQL文 (SELECT文)
  - StatementオブジェクトのexecuteQuery()メソッドを利用
- 更新系SQL文 (UPDATE・DELETE・INSERT文)
  - StatementオブジェクトのexecuteUpdate()メソッドを利用

# SELECT文

## 構文

SELECT カラム名 FROM テーブル名 WHERE 条件式

- 実行内容
  - 指定したテーブルにおいて条件式に一致するレコードを取り出し、指定されたカラムのデータのみを形式で取得する
- 各項目の指定
  - カラム名はカンマ「,」で区切って複数指定可能
  - カラム名に「\*」を指定するとすべてのカラム名を指定したのと同じ意味となる
  - WHEREを省略すると全レコードを取り出す
  - 条件式にはカラム名を使用できる
  - 文字列定数は一重引用符「'」を使用する
  - AND、ORなどの論理演算子を使用できる

# INSERT文

## 構文

```
INSERT INTO テーブル名 (カラム名) VALUES (設定値)
```

- 実行内容

- 指定したテーブルにレコードを追加する

- 各項目の指定

- (カラム名)と(設定値)はカンマ「,」で区切って、同数だけ指定でき、各カラムに対応する設定値を順に記載する
- (カラム名)は省略可能だが、その場合(設定値)にはデータベースに登録されている順に各カラムの値を指定する

# UPDATE文

## 構文

```
UPDATE テーブル名 SET カラム名=設定値 WHERE 条件式
```

- 実行内容
  - 指定したテーブルの中で条件に一致するレコードの指定したカラムの値を設定値に更新する
- 各項目の指定
  - 「カラム名=設定値」はカンマ「,」で区切って複数指定できる
  - WHERE以下を省略すると全レコードが更新対象となる

# DELETE文

## 構文

```
DELETE FROM テーブル名 WHERE 条件式
```

- 実行内容
  - 指定したテーブルから条件に合ったレコードを削除する
- 各項目の指定
  - WHERE以下を省略すると全レコードが削除される

# 参照系SQLの実行

- StatementオブジェクトのexecuteQuery()メソッドを利用
  - 引数 = 実行したいSQL文 (String)
  - 戻り値 = java.sql.ResultSetオブジェクト

## コード例

```
try {  
    ...  
    Statement stmt = conn.createStatement();  
    String query = "SELECT * FROM addrbk";  
    ResultSet rs = stmt.executeQuery(query);  
    ...  
} catch (SQLException ex) {  
    ex.printStackTrace();  
}
```

# 更新系SQLの実行

- StatementオブジェクトのexecuteUpdate()メソッドを利用
  - 引数 = 実行したいSQL文 (String)
  - 戻り値 = 更新行数 (int型)

## コード例

```
try {
    ...
    Statement stmt = conn.createStatement();
    String query = "INSERT INTO addrbk (ID,氏名,年齢)"
        + " VALUES (1, '鈴木', 43)";
    int count = stmt.executeUpdate(query);
    ...
} catch (SQLException ex) {
    ex.printStackTrace();
}
```

※ DDL (=Data Definition Language、CREATE TABLE文など)を実行する場合にもexecuteUpdateメソッドを使用(戻り値は0固定)

# SQLの実行結果

- executeQueryメソッドの戻り値
  - ResultSetオブジェクト
    - SQL文の参照結果を格納したテナオブジェクト
- executeUpdateメソッドの戻り値
  - int型の整数
    - SQL文の実行によって更新(挿入・削除)された行数

# ResultSetの使い方

- ResultSetとは
  - 問い合わせの結果に行単位でアクセスするためのコンテナオブジェクト
  - 結果の各データにアクセスするためのメソッドを持つ
    - next()メソッド・・・カーソルを移動する
    - getterメソッド・・・現在行の各フィールドの値を取得する

# ResultSet#next()メソッド(1)

- ResultSetはレコードごとにデータをアクセスする
  - 現在アクセスできるレコード位置を記憶している「カーソル」を持っている
  - ただし、ResultSet取得直後(初期状態)はカーソルはどの行も指していないことに注意

カーソル →

ID	氏名	年齢	...
1	田中	23	...
2	鈴木	34	...
3	山田	45	...
...	...	...	...

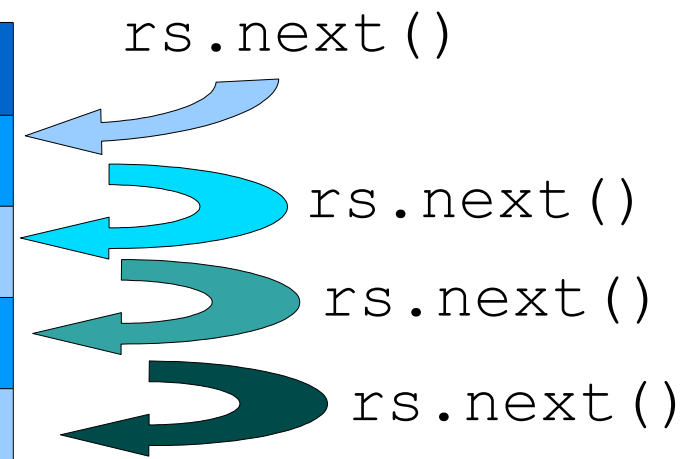
このレコードが現在アクセスできる

# ResultSet#next()メソッド(2)

- next()メソッドを実行するとカーソルが1行移動
  - 初期状態・・・先頭行にカーソルが移動
  - 上記以外・・・現在行の1行後ろへカーソルが移動

ID	氏名	年齢	...
1	田中	23	...
2	鈴木	34	...
3	山田	45	...
...	...	...	...

ResultSetオブジェクト(rs)

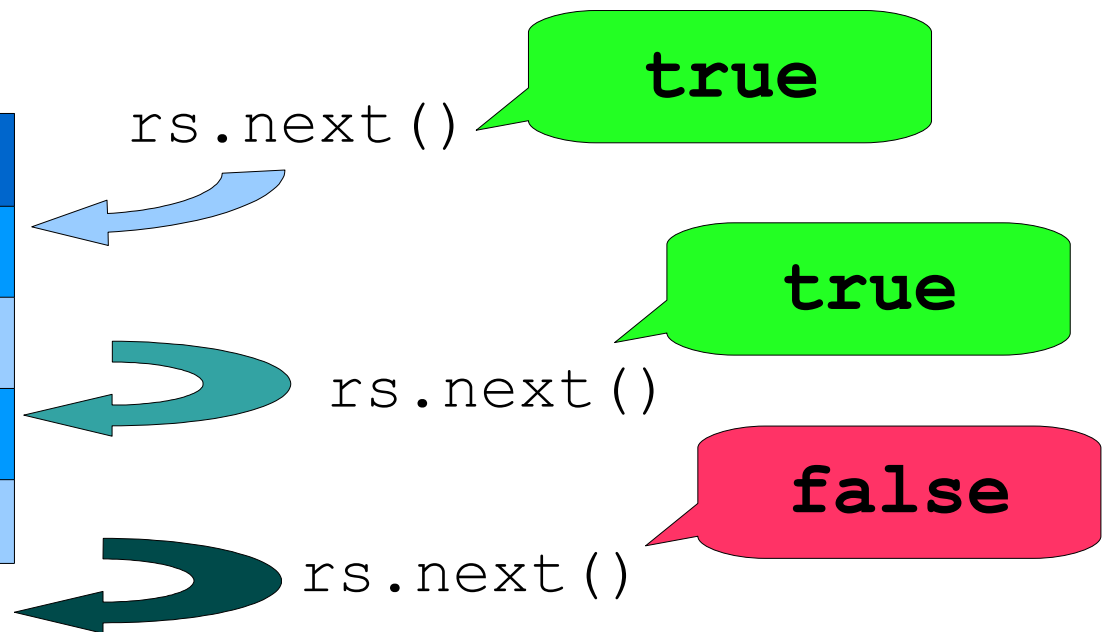


# ResultSet#next()メソッド(3)

- next()メソッドの戻り値
  - 後ろにレコードが存在するとき・・・「true」
  - もう後ろにレコードがないとき・・・「false」
- 戻り値をチェックすれば、カーソルが最終行に到達したかどうかをチェックできる

ID	氏名	年齢	...
1	田中	23	...
2	鈴木	34	...
3	山田	45	...
...	...	...	...

ResultSetオブジェクト(rs)



# getterメソッド(1)

- カーソルが指しているレコードのカラム値を取り出すためのメソッド群
  - getXXX()という名称 (XXXは型名)
    - getterメソッドの一例:
      - getString(), getInt(), getDate() など...
- 引数 = カラム名 (String) または カラム番号 (int)
- 戻り値 = フィールドの値 (型はメソッド名より異なる)

# getterメソッド(2)

- getterメソッド使用例

- rs.getInt(1); → カラム番号1の値を取得

- rs.getString("氏名"); → 「氏名」カラムの値を取得

## コード例

```
try {
    ...
    ResultSet rs = stmt.executeQuery(query);
    while(rs.next()) {
        int id = rs.getInt(1);
        String name = rs.getString("氏名");
        System.out.println(id+" "+name);
    }
    ...
} catch (SQLException ex) {
    ex.printStackTrace();
}
```

# getterメソッド一覧(抜粋)

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	BOOLEAN	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getBytes	◎	○	○	○	○	○	○	○	○	○	○	○	○	○						
getShort	○	◎	○	○	○	○	○	○	○	○	○	○	○	○						
getInt	○	○	◎	○	○	○	○	○	○	○	○	○	○	○						
getLong	○	○	○	◎	○	○	○	○	○	○	○	○	○	○						
getFloat	○	○	○	○	◎	○	○	○	○	○	○	○	○	○						
getDouble	○	○	○	○	○	◎	◎	○	○	○	○	○	○	○						
getBigDecimal	○	○	○	○	○	○	○	◎	◎	○	○	○	○	○						
getBoolean	○	○	○	○	○	○	○	○	○	◎	◎	○	○	○						
getString	○	○	○	○	○	○	○	○	○	○	○	◎	◎	○	○	○	○	○	○	○
getNString	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
getBytes															◎	◎	○			
getDate												○	○	○				◎		○
getTime												○	○	○					◎	○
getTimestamp												○	○	○				○	○	◎

注: Xは取得するJDBC型に対して推奨されるgetterメソッド

xはJDBC型に適応可能なgetterメソッド

参考:「JDBC™ 4.0 Specification」の「TABLE B-6 Type Conversions Supported by ResultSet  
getter Methods」より

# 接続の解除(1)

- データベースに対する各処理が終了したら接続を解除する
  - データベースへの接続は有限なため、不要となった接続は廃棄することが望ましい
  - ガベージコレクションによって、不要な接続は自動でクローズされるが、ガベージコレクションの実行タイミングはプログラムでは制御できないため、コード中で明示的に接続解除することが望ましい

# 接続の解除(2)

- 各クラス・インターフェースにあるclose()メソッドで接続を解除する

## コード例

```
Connection cnct;
Statement st;
ResultSet rs;
try {
    ...
} catch (SQLException ex) {
    ex.printStackTrace();
} finally {
    try {
        if (rs!=null) rs.close();
        if (st!=null) st.close();
        if (cnct!=null) cnct.close();
    } catch (Exception ex) {}
}
```

# JDBCアクセスコード完成例

```
String url = "jdbc:mysql://localhost/companydb";
String id = "myuser";
String pw = "mydata";
Connection cnct = null;
Statement st = null;
ResultSet rs = null;
try {
    Class.forName("com.mysql.jdbc.Driver");
    Connection cnct = DriverManager.getConnection(url, id, pw);
    Statement st = cnct.createStatement();
    ResultSet rs = st.executeQuery("SELECT * from addrbk");
    while(rs.next()){
        String name = rs.getString("氏名");
        String tel = rs.getString("電話番号");
        System.out.println("氏名:" + name + "\t電話番号:" + tel);
    }
} catch (ClassNotFoundException ex) {
    ex.printStackTrace();
} catch (SQLException ex) {
    ex.printStackTrace();
} finally {
    try {
        if (rs!=null) rs.close();
        if (st!=null) st.close();
        if (cnct!=null) cnct.close();
    } catch (Exception ex) { }
}
```

これらのコードを  
クラスのmainメソッド  
内などに記述します  
(import文も必要です)

# 演習問題(2)

- 下図のような構成のデータベースおよびテーブルがあります。
- これについて、次スライドの指示に従い演習を行ってください。

companydb  
データベース

addrbkテーブルの構造

カラム名	データ型	主キー
ID	INTEGER	○
NAME	VARCHAR	
GENDER	CHAR(1)	
AGE	INTEGER	
DEPT	VARCHAR	
ADDRESS	VARCHAR	
TEL	VARCHAR	

ID	NAME	GENDER	AGE	DEPT	ADDRESS	TEL
1	鈴木	女	43	総務部	東京都新宿区	03-1234-5678
2	田中	男	36	営業部	神奈川県横浜市	045-111-2222
3	佐藤	男	23	製造部	神奈川県川崎市	044-333-4444
4	加藤	男	53	研究開発部	埼玉県さいたま市	048-555-6666
5	山田	女	27	社長室	千葉県船橋市	047-777-8888

addrbkテーブル

# 演習問題(2)

(問1) 最初のレコードのカラム番号が5の列のデータを表示してください。

(問2) 最初のレコードのカラム名が「NAME」と「AGE」の列のデータを表示してください。

(問3) テーブルaddrbkからすべてのレコードを取り出して、名前(NAME)と電話番号(TEL)の一覧を表示してください。

# プリコンパイル

- プリコンパイルとは

- 同じ構成のSQL文で、値の内容だけを変えたものを繰り返しかえし実行するための方法
- あらかじめデータベースにSQL文を渡しておき、具体的な値を後から指定して実行する
- SQLを渡し、解析する手順が1度で済むので、繰り返しかえし実行する場合に効率が良い

# プリコンパイルの例

プリコンパイルさせるSQL例

```
INSERT INTO EMP (ID,NAME) VALUES (?,?)
```

データベースにSQLを渡し、あらかじめ解析させておく

適用したいパラメータ

(1, 'ナレッジ太郎')

SQL実行

適用したいパラメータ

(2, 'エックス二郎')

SQL実行

適用したいパラメータ

(3, 'KX三郎')

SQL実行

# JDBCによるプリコンパイル

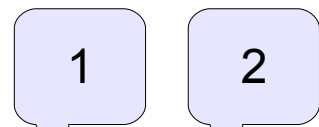
- プリコンパイルを実施するには
  - Statementインターフェースの代わりにPreparedStatementインターフェースを利用する
- 手順
  - PreparedStatementオブジェクトの取得
  - プリコンパイルしたいSQLを設定
  - 適用したいパラメータを設定
  - SQLを実行

# プリコンパイルしたいSQL

- パラメータの記述方法

- SQL中でパラメータとしたい部分は「?」で記述
- パラメータは複数記述可能
- SQL中での出現順に、番号が割り当てられる

パラメータ番号



プリコンパイルさせるSQL例

```
INSERT INTO EMP (ID,NAME) VALUES ( ?, ? )
```

パラメータ

# PreparedStatementの取得

- PreparedStatementオブジェクトの取得
  - Connection#prepareStatement()メソッドを利用
    - 引数 = プリコンパイルさせたいSQL文
    - 戻り値 = PreparedStatementオブジェクト

## コード例

```
try {  
    ...  
    Connection conn = DriverManager.getConnection(url, id, pw);  
    String query = "INSERT INTO EMP (ID,NAME) VALUES (?,?)";  
    PreparedStatement pstmt = conn.prepareStatement(query);  
    ...  
} catch (SQLException ex) {  
    ex.printStackTrace();  
}
```

# 適用したいパラメータの設定

- PreparedStatementオブジェクトに対し、setterメソッドでパラメータの値を設定できる
  - setterメソッド = 「set + 型名」のメソッド群の総称
  - 引数 = ①パラメータ番号、②パラメータの値

## コード例

```
try {
    ...
    String query = "INSERT INTO EMP (ID,NAME) VALUES (?,?)";
    PreparedStatement pstmt = conn.prepareStatement(query);
    pstmt.setInt(1,100);
    pstmt.setString(2,"ナレツジ四郎");
    ...
} catch (SQLException ex) {
    ex.printStackTrace();
}
```

パラメータ 1に「100」をセット

パラメータ 2に「ナレツジ四郎」をセット

# SQLの実行

- パラメータがセットできたら、executeQueryメソッドまたはexecuteUpdateメソッドでSQLを実行できる
  - 引数=なし
    - SQL文は既にオブジェクトに渡してあるため必要ない

## コード例

```
try {
    ...
    PreparedStatement pstmt = conn.prepareStatement(query);
    pstmt.setInt(1,100);
    pstmt.setString(2,"ナレツジ四郎");
    int count = pstmt.executeUpdate();
    ...
} catch (SQLException ex) {
    ex.printStackTrace();
}
```

# プリコンパイルの完成コード例

```
int newID[] = {6,7,8,9,10};
String newName[] = {"高橋","渡辺","伊藤","山本","中村"};
String url = "jdbc:mysql://localhost/companydb";
String id = "myuser";
String pw = "mydata";
Connection cnct = null;
PreparedStatement pst = null;
ResultSet rs = null;
try {
    Class.forName("com.mysql.jdbc.Driver");
    Connection cnct = DriverManager.getConnection(url,id,pw);
    String query = "INSERT INTO EMP (ID,NAME) VALUES (?,?)";
    PreparedStatement pst = cnct.prepareStatement(query);
    for(int i=0;i<newID.length;i++){
        pst.setInt(1, newID[i]);
        pst.setString(2, newName[i]);
        pst.executeUpdate();
    }
} catch(ClassNotFoundException ex){
    ex.printStackTrace();
} catch(SQLException ex) {
    ex.printStackTrace();
} finally {
    try {
        if (pst!=null) st.close();
        if (cnct!=null) cnct.close();
    } catch(Exception ex) { }
}
```

これらのコードを  
クラスのmainメソッド  
内などに記述します  
(import文も必要です)

# setterメソッド一覧(1)

引数(Javaの型)	メソッド名	JDBC SQL型
<code>java.sql.Array</code>	<code>setArray</code>	ARRAY
<code>java.io.InputStream</code>	<code>setAsciiStream</code>	LONGVARCHAR
<code>java.math.BigDecimal</code>	<code>setBigDecimal</code>	NUMERIC
<code>java.io.InputStream</code>	<code>setBinaryStream</code>	LONGVARBINARY
<code>java.sql.Blob</code>	<code>setBlob</code>	BLOB
<code>boolean</code>	<code>setBoolean</code>	BIT
<code>byte</code>	<code>setByte</code>	TINYINT
<code>byte[]</code>	<code>setBytes</code>	VARBINARY/ LONGVARBINARY
<code>java.io.Reader</code>	<code>setCharacterStream</code>	LONGVARCHAR
<code>java.sql.Clob</code>	<code>setClob</code>	CLOB
<code>java.sql.Date</code>	<code>setDate</code>	DATE
<code>double</code>	<code>setDouble</code>	DOUBLE
<code>float</code>	<code>setFloat</code>	REAL
<code>int</code>	<code>setInt</code>	INTEGER

# setterメソッド一覧(2)

引数 ( Javaの型 )	メソッド名	JDBC SQL型
long	setLong	BIGINT
java.io.Reader	setNCharacterStream	LONGVARCHAR
java.sql.NClob	setNClob	NCLOB
String	setNString	NCHAR/NVARCHAR
Object	setObject	対応したオブジェクト
java.sql.Ref	setRef	REF
java.sql.RowId	setRowId	ROWID
short	setShort	SMALLINT
java.sql.SQLXML	setSQLXML	SQLXML
String	setString	VARCHAR/ LONGVARCHAR
java.sql.Time	setTime	TIME
java.sql.Timestamp	setTimestamp	TIMESTAMP
java.net.URL	setURL	URL

(注) setNullメソッドは第2引数 java.sql.Typesで指定される型のSQL NULLを設定する

# 演習問題(3)

演習問題(2)と同じデータベース・テーブルに対して、以下のプログラムを作成してください。

(問1) INSERT文をプリコンパイルして、以下のデータを追加してください(ID,NAME以外のカラムは指定しなくてよい)。

ID	NAME
6	高橋
7	渡辺
8	伊藤
9	山本
10	中村

(問2) DELETE文をプリコンパイルして、(問1)で追加したID=6～10のデータを削除してください。

# トランザクション(1)

- トランザクションとは
  - データベースの更新処理において「分離できない一連の処理」のこと
  - 「分離できない一連の処理」の例
    - 銀行口座の振込処理
      - ①ある口座の残高を減らす
      - ②ある口座の残高を増やす
      - どちらかが欠けても振込処理は成立しない

# トランザクション(2)

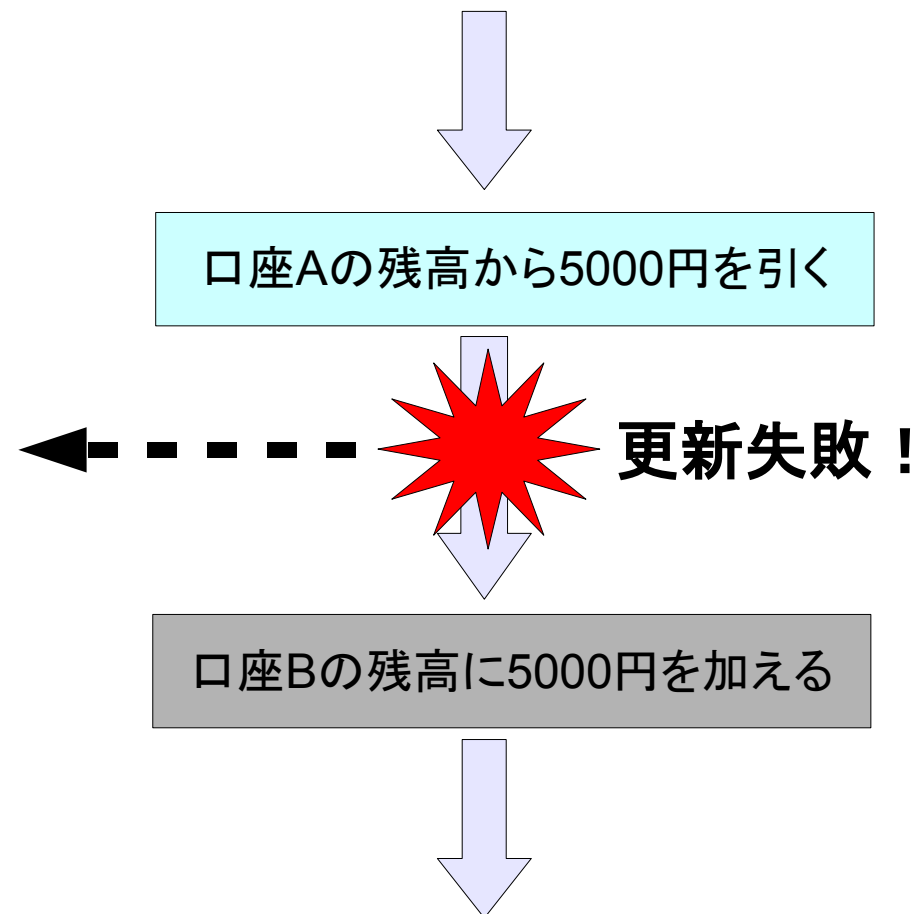
- 例：口座Aから口座Bに5000円を振り込む処理

口座番号	口座名義	預金残高
1234567	A	500,000
1234568	B	1,200,000
1234569	C	250,000

預金口座テーブル

ここで処理が中断されると、  
口座Aからの引き落としだけが  
実行され、口座Bの預金額  
が増えないままになってしまう

データの不整合



# トランザクション(3)

- コミットとロールバック

- コミット(commit)

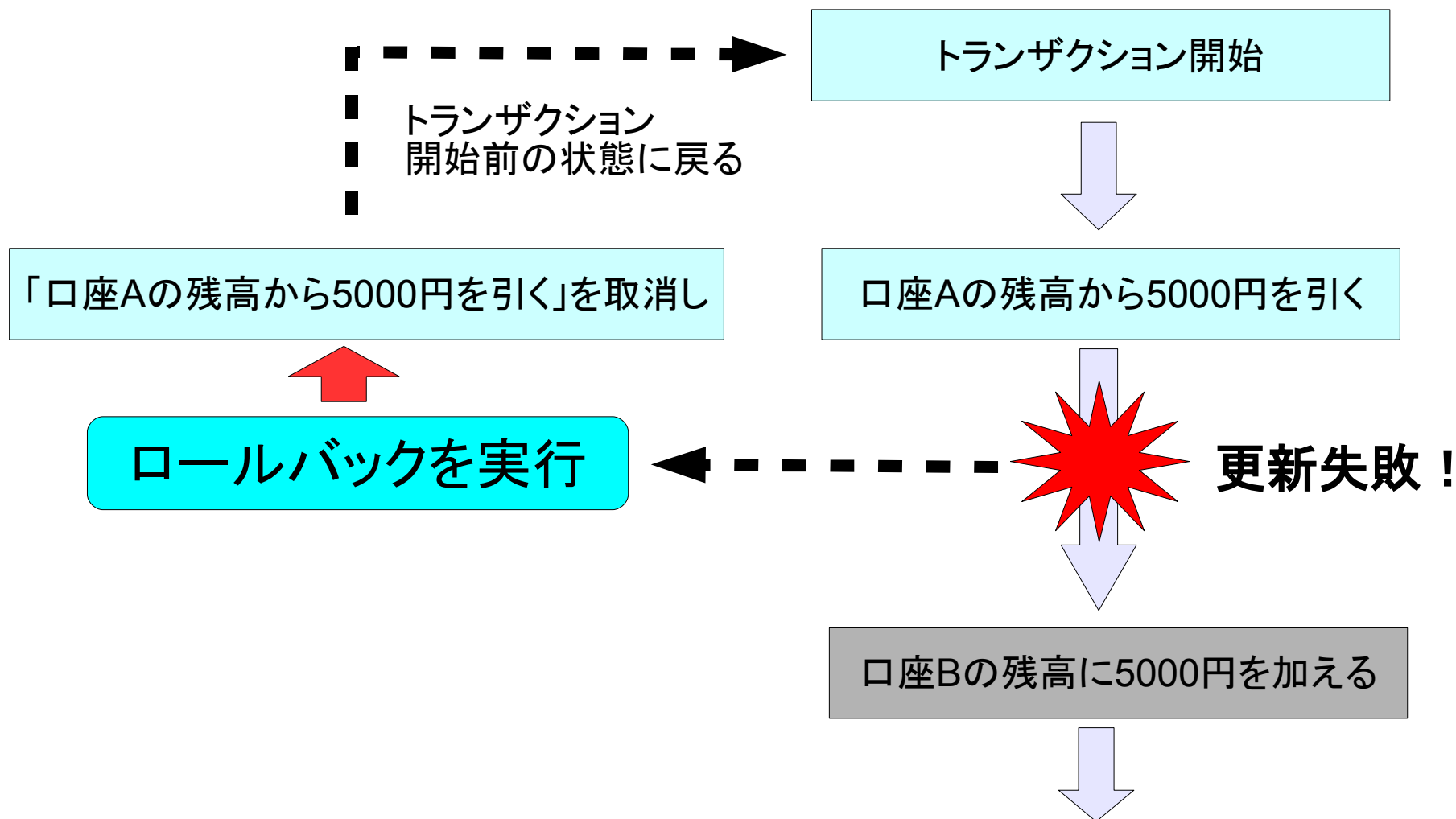
- トランザクションに含まれる一連の処理を仮実行しておき、中断することなく実行できたときに、これをデータベースに確定する処理

- ロールバック(rollback)

- トランザクションに含まれる一連の処理の仮実行中に失敗があったとき、トランザクションの全ての処理をキャンセルして元の状態に戻す処理

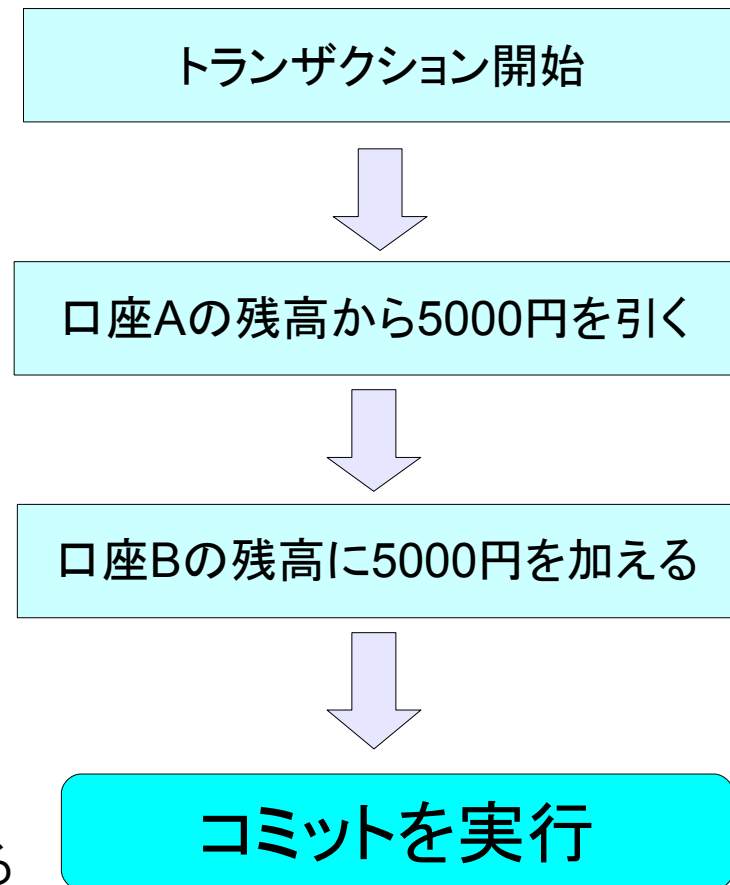
# トランザクション(4)

- ロールバックの例



# トランザクション(4)

- コミットの例



トランザクション内の  
更新内容が確定される

# JDBCのトランザクション(1)

- JDBCのデフォルト動作
  - 実行ごとにコミットが発行される「自動コミットモード」
  - 複数の更新処理をトランザクションとして扱いたい場合は「自動コミットモード」を解除する
- 自動コミットモードの解除方法
  - Connection#setAutoCommit()メソッド
    - 引数=false(モード解除)
    - 戻り値=なし

## コード例

```
...  
Connection conn = DriverManager.getConnection(url, id, pass);  
conn.setAutoCommit(false);  
...
```

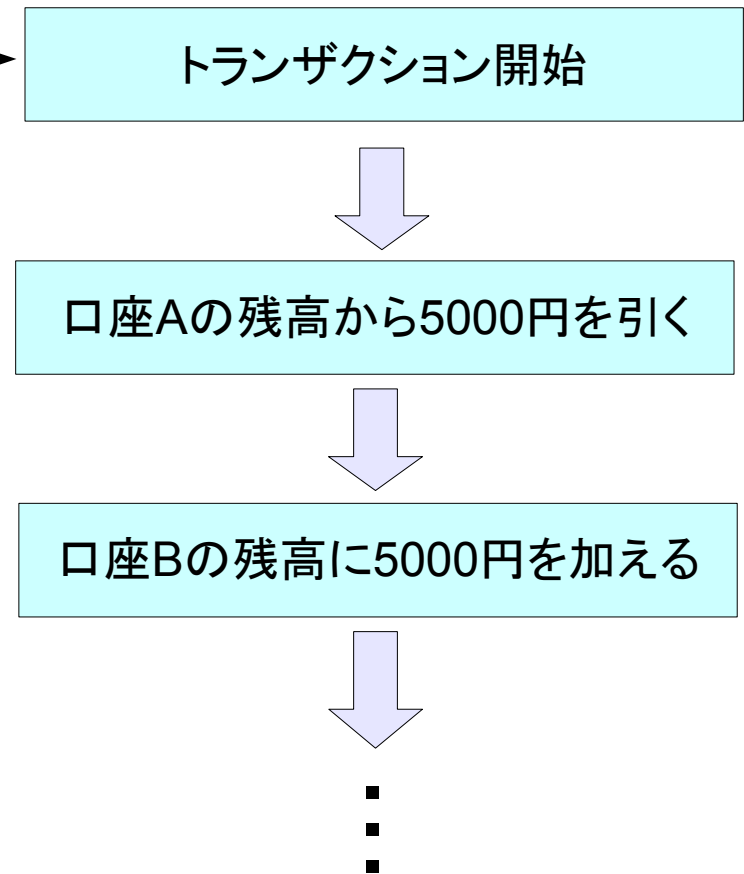
# JDBCのトランザクション(2)

## • 自動コミットモードの解除

```
conn.setAutoCommit(false);
```

自動コミットモードが解除されると、  
そこから後の更新処理がトランザクション  
として扱われる

次にコミットまたはロールバックを実行すると  
それ以降の更新処理が新たなトランザクション  
として扱われるため、自動コミットモードの解除  
は最初に一度だけの実行でよい



# JDBCのトランザクション(3)

- Connection#commit()メソッド
  - コミットを発行しデータベース内容を確定させる
  - 引数・戻り値=なし

## コード例

```
...  
Connection conn = DriverManager.getConnection(url, id, pass);  
conn.setAutoCommit(false);  
Statement st = conn.createStatement();  
String query1 = "UPDATE ACCOUNT SET MONEY=4000 WHERE ID=1234567";  
String query2 = "UPDATE ACCOUNT SET MONEY=6000 WHERE ID=1234568";  
st.executeUpdate(query1);  
st.executeUpdate(query2);  
conn.commit();  
...
```

# JDBCのトランザクション(4)

- Connection#rollback()メソッド
  - ロールバックを発行し更新内容を破棄する
  - 引数・戻り値=なし

## コード例

```
try {  
    ...  
    conn.setAutoCommit(false);  
    Statement st = conn.createStatement();  
    String query1 = "UPDATE ACCOUNT SET MONEY=4000 WHERE ID=1234567";  
    String query2 = "UPDATE ACCOUNT SET MONEY=6000 WHERE ID=1234568";  
    st.executeUpdate(query1);  
    st.executeUpdate(query2);  
    conn.commit();  
} catch (SQLException ex) {  
    conn.rollback(); ◀  
}
```

更新が失敗した場合はロールバックする

# 演習問題(4)

演習問題(2)と同じデータベース・テーブルに対して、以下のプログラムを作成してください。

(問1)

演習問題(3)のデータの追加をトランザクションを利用して更新し、全データを例外を発生することなく更新したらコミットし、例外が発生したらロールバックしてください。

(問2)

上記(問1)のプログラムと同様に右表のデータを追加するプログラムを作成してください。このプログラムを実行するとロールバックがおこり、データが追加されないことを確認してください。(主キーの重複により例外が発生する)

ID	NAME
11	佐々木
12	斎藤
5	山口
13	松本

# メタデータクラス

- データベーステーブルのカラム名や型、カラム数などの情報を取得するためのインターフェース群
  - DatabaseMetaDataインターフェース
    - データベースの情報を取得するためのインターフェース
    - スキーマ名、カタログ名、テーブルの列名、データ型、SQLのサポートレベル、製品名、ドライバ情報など
    - Connection#getMetaData()でオブジェクト取得可能
  - ResultSetMetaDataインターフェース
    - ResultSetの情報を取得するためのインターフェース
    - カラム数、カラム名、データ型など
    - ResultSet.getMetaData()でオブジェクト取得可能

# 付録: JDBCのバージョン

- JDBC 1.0
  - 1997年1月に制定。SQLデータベースへの基本呼び出しレベルのインターフェイスのみ。
- JDBC 2.1/2.0オプションパッケージ
  - アプリケーションからJDBC APIの使用を管理するアプリケーションサーバに必要な機能をサポート。
- JDBC 3.0
  - 2.0ではカバーしきれていなかった、僅かに不足する機能を補い、APIを完成させるために制定。
- JDBC 4.0
  - 開発しやすさを向上させ、JDBCリソースを管理する豊富な機能を持ったツールとAPIにより企業レベルの使用に耐えうる機能を提供。