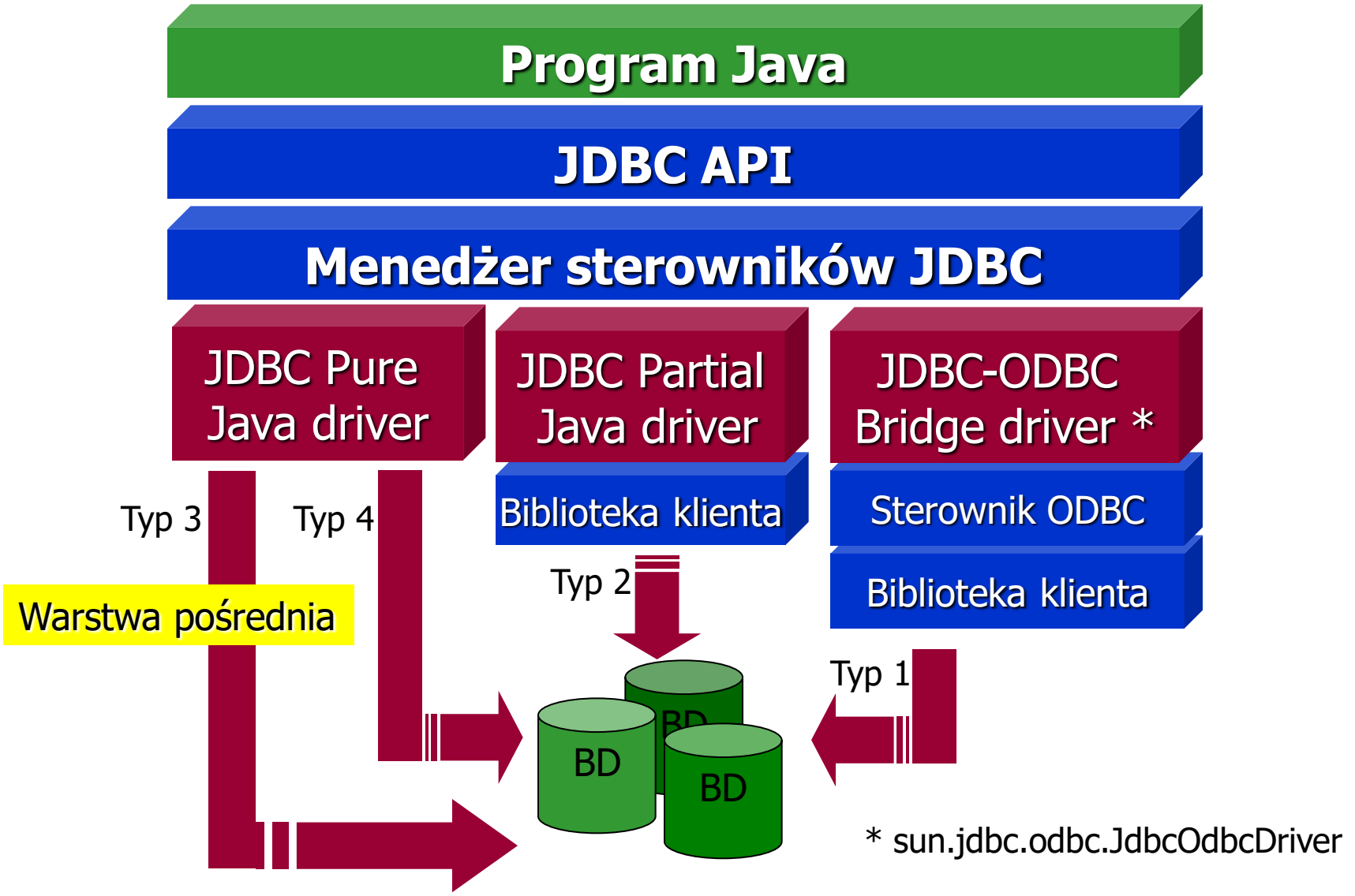
# JDBC
# (Java Database Connectivity)

Marek Wojciechowski

# Czym jest JDBC ?

- JDBC jest standardowym interfejsem do współpracy aplikacji Java z relacyjną bazą danych
- JDBC definiuje standardowe interfejsy
  - interfejsy są implementowane przez sterowniki JDBC
  - standard dopuszcza rozszerzenia producentów
- Zaprojektowany na podstawie X/Open SQL Call Level Interface (podobny do ODBC)
- Interfejs dla dynamicznych instrukcji SQL
- Opracowany przez Sun Microsystems
- Powszechnie wspierany przez producentów systemów baz danych i narzędzi programistycznych

# Architektura JDBC



# Typy sterowników JDBC

- Typ I – Most JDBC-ODBC
  - umożliwia połączenie z każdą bazą danych, dla której istnieje sterownik ODBC
- Typ II – Sterownik napisany częściowo w Javie, wykorzystujący biblioteki klienta bazy danych
  - najbardziej wydajne rozwiązanie (szczególnie dla starszych JVM)
  - wymaga preinstalowanego oprogramowania klienta bazy danych
- Typ III – Uniwersalny sterownik w czystej Javie, z obsługą specyficznych baz danych w warstwie pośredniej
  - Oprogramowanie klasy middleware (serwer aplikacji) tłumaczy uniwersalne wywołania JDBC na protokoły specyficzne dla poszczególnych serwerów b.d.
- Typ IV – Sterownik w czystej Javie, komunikujący się bezpośrednio z serwerem bazy danych
  - nie wymaga bibliotek klienta bazy danych

# Sterowniki JDBC Oracle

`oracle.jdbc.OracleDriver`

- JDBC Thin (typ IV)
  - w 100% napisany w czystej Javie
- JDBC OCI (typ II)
  - wykonuje wywołania OCI do "fabrycznego" sterownika, preinstalowanego po stronie klienta
- Server-side internal driver
  - wykorzystywany przez aplikacje Java uruchamiane wewnątrz serwera Oracle (np. Java Stored Procedures)
- Server-side Thin driver
  - wykorzystywany przez aplikacje Java uruchamiane wewnątrz serwera Oracle do nawiązywania połączeń z innymi serwerami

# Podstawowe klasy JDBC (pakiet java.sql)

- `DriverManager`
- `Connection`
- `Statement`
- `PreparedStatement`
- `CollableStatement`
- `ResultSet`
- `DatabaseMetaData`
- `ResultSetMetaData`
- `SQLException`

# Rejestrowanie sterowników JDBC

- Sterowniki JDBC muszą się rejestrować w menedżerze sterowników (**`DriverManager`**)
- Sterowniki rejestrują się automatycznie podczas ich ładowania przez JVM (**`Class.forName`**())
- Od JDBC 4.0 JVM automatycznie ładuje sterowniki z dostępnych bibliotek

```
try {
  // Oracle JDBC driver (Thin + OCI)
  Class.forName("oracle.jdbc.OracleDriver");
  // IBM DB2 Universal Database ”app” driver
  Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
  // JDBC-ODBC bridge driver
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  // Java DB (derby) driver
  Class.forName("org.apache.derby.jdbc.ClientDriver");
}
catch (ClassNotFoundException e) { ... }
```

# Nawiązywanie połączenia z bazą danych

- Menedżer sterowników (klasa **`DriverManager`**) zarządza sterownikami JDBC i służy do otwarcia połączenia z bazą danych
- Baza danych jest wskazywana przez podanie JDBC URL, identyfikującego sterownik JDBC i bazę danych:

  **`jdbc:<subprotocol>:<connectString>`**
- Na podstawie JDBC URL menedżer sterowników (**`DriverManager`**) wybiera odpowiedni sterownik JDBC spośród zarejestrowanych sterowników

# Formaty JDBC URL - Przykłady

| | |
|---|---|
| JDBC-ODBC Bridge | `jdbc:odbc:<odbc_data_source>` |
| Oracle OCI | `jdbc:oracle:oci:@<service_name>` |
| Oracle Thin | `jdbc:oracle:thin:@<host>:<port>:<sid>` |
| IBM DB2 ”app” | `jdbc:db2:<db_name>` |
| IBM DB2 ”net” | `jdbc:db2://<host>:<port>/<db_name>` |
| MySQL | `jdbc:mysql://<host>:<port>/<db_name>` |
| Java DB (Derby) | `jdbc:derby://<host>:<port>/<db_name>` |

# Wyjątek SQLException

- Wywołania JDBC mogą generować wyjątek **`java.sql.SQLException`**
- Metody zawierające wywołania JDBC muszą obsługiwać ten wyjątek lub deklarować możliwość jego generowania
- Wyjątek **`SQLException`** niesie następujące informacje:
  - kod "SQL state" (zgodny ze specyfikacją XOPEN SQL)
  - tekstowy komunikat o błędzie
  - numeryczny kod błędu (specyficzny dla danego DBMS)

```
try {
   conn = DriverManager.getConnection("jdbc:odbc:Finance",
                                      "scott","tiger");
} catch (SQLException e) {
  System.err.println("Error: " + e);
  String sqlState = e.getSQLState();
  String message = e.getMessage();
  int errorCode = e.getErrorCode();
  ...
}
```

# Polecenia SQL w JDBC

- Polecenie **`Statement`**
  - wykonywanie zapytań lub operacji DML/DDL
- Polecenie **`PreparedStatement`** (wywiedzione z **`Statement`**)
  - wykonywanie poleceń prekompilowanych
  - możliwość zaszycia zmiennych
  - przydatne gdy to samo polecenie jest wykonywane kilkukrotnie dla różnych wartości
  - zwiększa odporność na ataki „SQL injection”
- Polecenie **`CallableStatement`** (wywiedzione z **`PreparedStatement`**)
  - wywoływanie procedur i funkcji składowanych w bazie danych
  - zachowana możliwość zaszywania zmiennych

# Klasa Statement

- Wykonywanie zapytań (metoda **`executeQuery()`** zwracająca zbiór wynikowy **`ResultSet`**)

```
try {
  Statement stmt = conn.createStatement();
  ResultSet rset = stmt.executeQuery
              ("SELECT name, salary FROM employees");
...
} catch (SQLException e) {…}
```

- Wykonywanie poleceń DML i DDL (metoda **`executeUpdate()`**)

```
try {
  Statement stmt = conn.createStatement();
  stmt.executeUpdate
            ("DELETE FROM employees WHERE id = 120");
} catch (SQLException e) {…}
```

# Przetwarzanie zbiorów wynikowych

- Obiekt **`ResultSet`** przechowuje tabelę danych wynikowych zwróconych przez zapytanie SQL
- Obsługa kursorów
  - kursor startuje od pozycji przed pierwszym rekordem zbioru wynikowego
  - metoda **`next()`** przesuwa kursor na następny rekord (zwraca **`true`**, gdy znaleziono kolejny rekord)
  - od JDBC 2.0 są przewijalne i modyfikowalne zbiory wynikowe
- Dane odczytuje się przy pomocy metod **`getXXX()`** np. **`getInt()`**, **`getString()`**, które odwzorowują wyniki w równoważne typy języka Java
- Dostęp do pól bieżącego rekordu odbywa się przez numer na liście wyrażeń w zapytaniu lub nazwę atrybutu

# Przetwarzanie zbiorów wynikowych - Przykłady

```
try {
  ResultSet rset = stmt.executeQuery(
                   "SELECT name, salary FROM employees");
  while (rset.next()) {
    String name = rset.getString(1);
    int salary = rset.getInt(2);
  }
} catch (SQLException e) {…}
```

```
try {
  ResultSet rset = stmt.executeQuery(
                   "SELECT name, salary FROM employees");
  while (rset.next()) {
    String name = rset.getString("NAME");
    int salary = rset.getInt("SALARY");
  }
} catch (SQLException e) {…}
```

# JDBC – kompletny przykład

```
import java.sql.*;

public class MyDemo {
   public static void main(String[] args)
   throws ClassNotFoundException, SQLException {
     Connection conn;
     Statement stmt;
     ResultSet rset;

     Class.forName("oracle.jdbc.OracleDriver");
     conn = DriverManager.getConnection(
        "jdbc:oracle:thin:@srv1:1521:orcl","scott","tiger");
     stmt = conn.createStatement();
     rset = stmt.executeQuery("select name, salary from employees");
     while (rset.next()) {
        System.out.print(rset.getString("NAME")+" ");
        System.out.println(rset.getString("SALARY"));
     }
     rset.close(); stmt.close(); conn.close();
}}
```

# JDBC 4.1 – zarządzanie zasobami

```
try (Statement stmt = con.createStatement()) {

      ResultSet rs = stmt.executeQuery(query);

      while (rs.next()) {
        ... = rs.getXXX("...");
        ...
      }

    } catch (SQLException e) {
      ...
    }
```

- Opuszczenie bloku **`try`** (normalne zakończenie lub wystąpienie wyjątku) spowoduje automatyczne zamknięcie obiektu polecenia (**`Statement`**)
- Zamknięcie polecenia powoduje zamknięcie zbiorów wynikowych (**`ResultSet`**) z nim związanych
- Analogicznie można obsłużyć zamknięcie połączenia (**`Connection`**)

# Konwersje typów między SQL i Java

- Dla każdego typu SQL istnieje jeden lub więcej typów Java, do których możliwa jest konwersja:

```
CHAR -> String, int, double, float, java.math.BigDecimal, ...
NUMBER -> int, double, float, java.math.BigDecimal, ...
DATE -> Date, Time, Timestamp, String
...
```

- Konwersja z SQL do Java może wiązać się z utratą precyzji
- Producenci systemów baz danych mogą dostarczać typy Java pozwalające na uniknięcie utraty precyzji przy konwersji z SQL (przykładem jest Oracle)
- Specyficzne typy Java do obsługi typów Oracle SQL:

```
CHAR -> oracle.sql.CHAR
NUMBER -> oracle.sql.NUMBER
DATE -> oracle.sql.DATE
...
```

- Do konwersji do typów **`oracle.sql.*`** służą odpowiednie metody **`getXXX()`** np. **`getCHAR()`**

# Klasa PreparedStatement

- Umożliwiają wielokrotne wykonanie tego samego polecenia dla różnych wartości parametrów
  - wydajność
- Parametry wstawia się za pomocą znaku „?”
- Zapobieganie atakom „SQL injection”

```
try {
  Connection conn = DriverManager.getConnection(…);

  PreparedStatement pstmt =
    conn.prepareStatement(
     "UPDATE employees SET salary = 9000 WHERE id = ?");
  …
} catch (SQLException e) {…}
```

# Wiązanie zmiennych i wykonywanie polecenia PreparedStatement

- Przed wywołaniem polecenia **`PreparedStatement`** należy związać parametry z konkretnymi wartościami
- Parametry wiąże się metodami **`setXXX()`** klasy **`PreparedStatement`**

```
try {
  PreparedStatement pstmt =
    conn.prepareStatement(
      "UPDATE employees SET salary = 8900 WHERE id = ?");
  …
  pstmt.setInt(1, 120);
  pstmt.executeUpdate();

  pstmt.setInt(1, 130);
  pstmt.executeUpdate();
  …
} catch (SQLException e) {…}
```

# Transakcje

- Przetwarzanie transakcyjne zależy od właściwości **`autoCommit`** obiektu **`Connection`**
  - domyślnie **`true`**, co oznacza oddzielną transakcję dla każdego polecenia SQL (każda instrukcja jest automatycznie zatwierdzana)
  - do zmiany trybu służy metoda **`setAutoCommit()`**
  - gdy **`autoCommit`** == **`false`**:
    - **`commit()`** - zatwierdzenie transakcji
    - **`rollback()`** - wycofanie transakcji
    - **`setSavepoint()`** - ustawienie punktu bezpieczeństwa

```
Connection conn = DriverManager.getConnection(…);
conn.setAutoCommit(false);  // zmiana trybu
...                            // polecenia SQL
conn.commit();              // zatwierdzenie
...                            // polecenia SQL
conn.rollback();            // wycofanie transakcji
```

# Źródła danych (DataSources)

- Obiekt **`DataSource`** reprezentuje źródło danych (np. DBMS)
- Obiekt **`DataSource`** w zakresie uzyskiwania połączenia z bazą danych jest alternatywą dla **`DriverManager`**
  - przede wszystkim w aplikacjach Java EE (na serwerze aplikacji)
- Źródło **`DataSource`** może wspierać:
  - transakcje rozproszone
  - obsługę pul połączeń
- Źródło **`DataSource`** może zostać zarejestrowane w JNDI
- Aplikacja łącząc się z bazą danych:
  - wyszukuje źródło danych w JNDI przez jego logiczną nazwę
  - poprzez obiekt **`DataSource`** uzyskuje obiekt **`Connection`**
- Połączenie uzyskane poprzez **`DataSource`** może brać udział w transakcjach rozproszonych i/lub funkcjonować w ramach puli połączeń zależnie od właściwości źródła

# Definicja źródła danych – Przykład

- Przykład dla serwera GlassFish

domain.xml

```
<resources>
   <jdbc-resource enabled="true" jndi-name="jdbc/sample"
                  object-type="user" pool-name="SamplePool"/>
   <jdbc-connection-pool
         ...
         datasource-classname="org.apache.derby.jdbc.ClientDataSource"
         max-pool-size="32"
         name="SamplePool"
         res-type="javax.sql.DataSource"
         ...
         >
     <property name="DatabaseName" value="sample"/>
     <property name="User" value="app"/>
     <property name="Password" value="app"/>
     <property name="URL" value="jdbc:derby://localhost:1527/sample"/>
     <property name="PortNumber" value="1527"/>
     <property name="serverName" value="localhost"/>
   </jdbc-connection-pool>
 </resources>
```

# Korzystanie ze źródeł danych – Przykład

- Dostęp do źródła poprzez jawne wyszukanie w serwisie nazw JNDI (ang. lookup)

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/sample");
Connection conn = ds.getConnection();
...
```

- Dostęp do źródła dzięki wstrzykiwaniu zależności (ang. dependency injection)

```
@Resource(name="jdbc/sample")
private DataSource ds;

Connection conn = ds.getConnection();
...
```