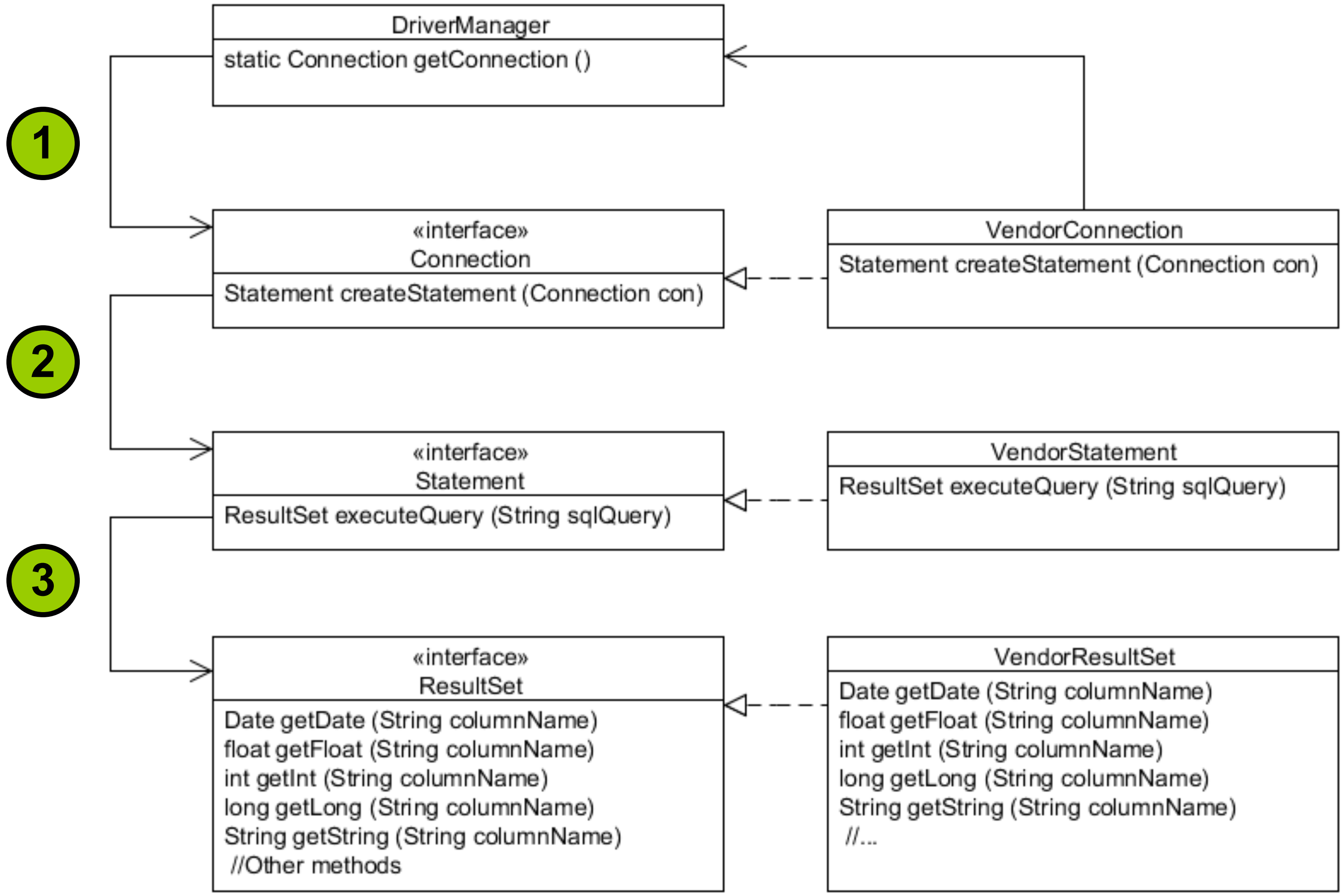
14

Building Database Applications with JDBC

Using the JDBC API


DriverManager
static Connection getConnection ()
1
«interface»
Connection
Statement createStatement (Connection con)
VendorConnection
Statement createStatement (Connection con)
2
«interface»
Statement
ResultSet executeQuery (String sqlQuery)
VendorStatement
ResultSet executeQuery (String sqlQuery)
3
«interface»
ResultSet
Date getDate (String columnName)
float getFloat (String columnName)
int getInt (String columnName)
long getLong (String columnName)
String getString (String columnName)
//Other methods
VendorResultSet
Date getDate (String columnName)
float getFloat (String columnName)
int getInt (String columnName)
long getLong (String columnName)
String getString (String columnName)
//...

Using a Vendor's Driver Class

The `DriverManager` class is used to get an instance of a Connection object, using the JDBC driver named in the JDBC URL:

```
String url = "jdbc:derby://localhost:1527/EmployeeDB";
Connection con = DriverManager.getConnection (url);
```

- The URL syntax for a JDBC driver is:

```
jdbc:<driver>:[subsubprotocol:][databaseName][;attribute=value]
```

- Each vendor can implement its own subprotocol.
- The URL syntax for an Oracle Thin driver is:

```
jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE
```

Example:

```
jdbc:oracle:thin:@//myhost:1521/orcl
```

Key JDBC API Components

Each vendor's JDBC driver class also implements the key API classes that you will use to connect to the database, execute queries, and manipulate data:

- `java.sql.Connection`: A connection that represents the session between your Java application and the database

```
Connection con = DriverManager.getConnection(url,
    username, password);
```

- `java.sql.Statement`: An object used to execute a static SQL statement and return the result

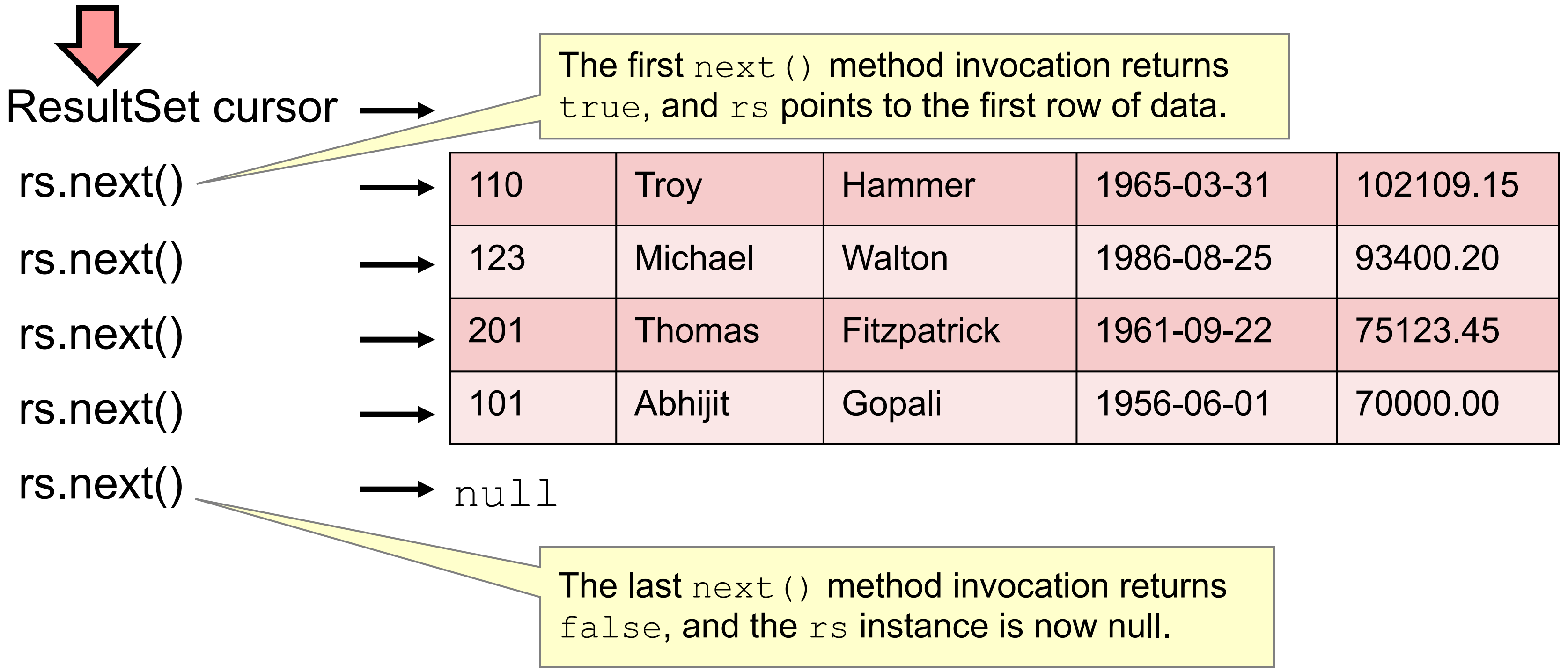
```
Statement stmt = con.createStatement();
```

- `java.sql.ResultSet`: A object representing a database result set

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```

Using a `ResultSet` Object

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```



The first `next()` method invocation returns `true`, and `rs` points to the first row of data.

110	Troy	Hammer	1965-03-31	102109.15
123	Michael	Walton	1986-08-25	93400.20
201	Thomas	Fitzpatrick	1961-09-22	75123.45
101	Abhijit	Gopali	1956-06-01	70000.00

The last `next()` method invocation returns `false`, and the `rs` instance is now null.

Putting It All Together

```java
package com.example.text;

import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Date;

public class SimpleJDBCTest {

    public static void main(String[] args) {
        String url = "jdbc:derby://localhost:1527/EmployeeDB";
        String username = "public";
        String password = "tiger";
        String query = "SELECT * FROM Employee";
        try (Connection con =
             DriverManager.getConnection (url, username, password);
             Statement stmt = con.createStatement ();
             ResultSet rs = stmt.executeQuery (query)) {
```

The hard-coded JDBC URL, username, and password is just for this simple example.

Putting It All Together

Loop through all of the rows in the `ResultSet`.

```
            while (rs.next()) {
                int empID = rs.getInt("ID");
                String first = rs.getString("FirstName");
                String last = rs.getString("LastName");
                Date birthDate = rs.getDate("BirthDate");
                float salary = rs.getFloat("Salary");
                System.out.println("Employee ID:   " + empID + "\n"
                + "Employee Name: " + first + " " + last + "\n"
                + "Birth Date:    " + birthDate + "\n"
                + "Salary:        " + salary);
            } // end of while
        } catch (SQLException e) {
            System.out.println("SQL Exception: " + e);
        } // end of try-with-resources
    }
}
```

Writing Portable JDBC Code

The JDBC driver provides a programmatic “insulating” layer between your Java application and the database. However, you also need to consider SQL syntax and semantics when writing database applications.

- Most databases support a standard set of SQL syntax and semantics described by the American National Standards Institute (ANSI) SQL-92 Entry-level specification.
- You can programmatically check for support for this specification from your driver:

```
Connection con = DriverManager.getConnection(url, username,
   password);
DatabaseMetaData dbm = con.getMetaData();
if (dbm.supportsANSI92EntrySQL()) {
   // Support for Entry-level SQL-92 standard
}
```

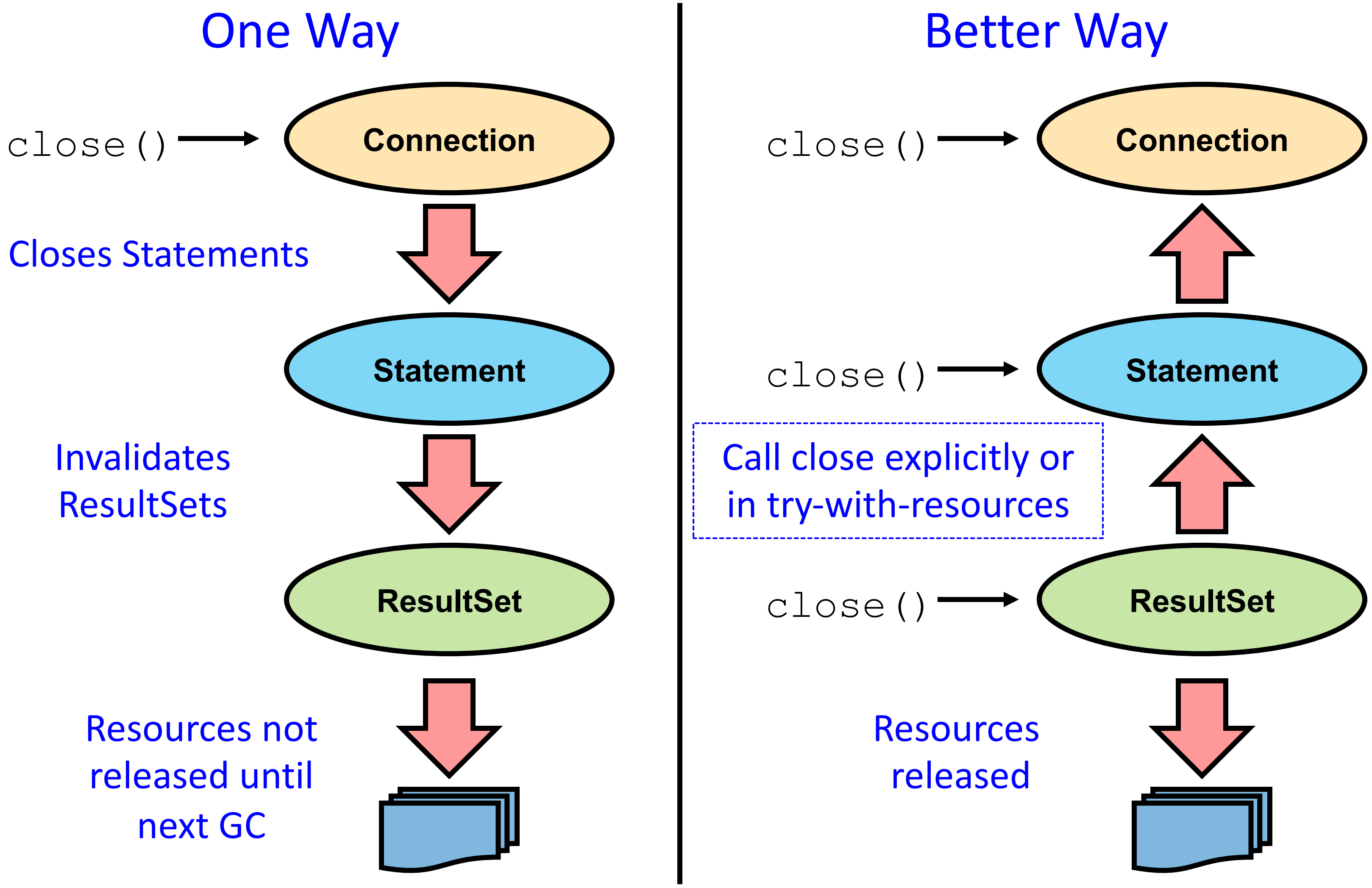
The SQLException Class

SQLException can be used to report details about resulting database errors. To report all the exceptions thrown, you can iterate through the SQLExceptions thrown:

```
catch(SQLException ex) {
    while(ex != null) {
        System.out.println("SQLState:  " + ex.getSQLState());
        System.out.println("Error Code:" + ex.getErrorCode());
        System.out.println("Message:   " + ex.getMessage());
        Throwable t = ex.getCause();
        while(t != null) {
            System.out.println("Cause:" + t);
            t = t.getCause();
         }
         ex = ex.getNextException();
    }
}
```

Vendor-dependent state codes, error codes and messages

Closing JDBC Objects



The `try`-with-resources Construct

Given the following `try`-with-resources statement:

```
try (Connection con =
     DriverManager.getConnection(url, username, password);
     Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery (query)){
```

- The compiler checks to see that the object inside the parentheses implements `java.lang.AutoCloseable`.
  - This interface includes one method: `void close()`.
- The close method is automatically called at the end of the try block in the proper order (last declaration to first).
- Multiple closeable resources can be included in the try block, separated by semicolons.

`try`-with-resources: Bad Practice

It might be tempting to write `try`-with-resources more compactly:

```
try (ResultSet rs = DriverManager.getConnection(url, username,
password).createStatement().executeQuery(query)) {
```

- However, only the close method of `ResultSet` is called, which is not a good practice.
- Always keep in mind which resources you need to close when using `try`-with-resources.

Writing Queries and Getting Results

To execute SQL queries with JDBC, you must create a SQL query wrapper object, an instance of the Statement object.

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

- Use the Statement instance to execute a SQL query:

```
ResultSet rs = stmt.executeQuery (query);
```

- Note that there are three Statement execute methods:

Method	Returns	Used for
`executeQuery(sqlString)`	`ResultSet`	`SELECT` statement
`executeUpdate(sqlString)`	`int` (rows affected)	`INSERT`, `UPDATE`, `DELETE`, or a DDL
`execute(sqlString)`	`boolean` (true if there was a `ResultSet`)	Any SQL command or commands

ResultSetMetaData

There may be a time where you need to dynamically discover the number of columns and their type.

```
int numCols = rs.getMetaData().getColumnCount();
String [] colNames = new String[numCols];
String [] colTypes = new String[numCols];
for (int i= 0; i < numCols; i++) {
    colNames[i] = rs.getMetaData().getColumnName(i+1);
    colTypes[i] = rs.getMetaData().getColumnTypeName(i+1);
}
System.out.println ("Number of columns returned: " + numCols);
System.out.println ("Column names/types returned: ");
for (int i = 0; i < numCols; i++) {
    System.out.println (colNames[i] + " : " + colTypes[i]);
}
```

Note that these methods are indexed from 1, not 0.

Using `PreparedStatement`

`PreparedStatement` is a subclass of Statement that allows you to pass arguments to a precompiled SQL statement.

Parameter for substitution.

```
double value = 100_000.00;
String query = "SELECT * FROM Employee WHERE Salary > ?";
PreparedStatement pStmt = con.prepareStatement(query);
pStmt.setDouble(1, value);
ResultSet rs = pStmt.executeQuery();
```

Substitutes `value` for the first parameter in the prepared statement.

- In this code fragment, a prepared statement returns all columns of all rows whose salary is greater than $100,000.
- `PreparedStatement` is useful when you have a SQL statements that you are going to execute multiple times.

Using `CallableStatement`

A `CallableStatement` allows non-SQL statements (such as stored procedures) to be executed against the database.

```
CallableStatement cStmt
       = con.prepareCall("{CALL EmplAgeCount (?, ?)}");
int age = 50;
cStmt.setInt (1, age);
ResultSet rs = cStmt.executeQuery();
cStmt.registerOutParameter(2, Types.INTEGER);
boolean result = cStmt.execute();
int count = cStmt.getInt(2);
System.out.println("There are " + count +
          " Employees over the age of " + age);
```

The `IN` parameter is passed in to the stored procedure.

The `OUT` parameter is returned from the stored procedure.

- Stored procedures are executed on the database.

What Is a Transaction?

- A transaction is a mechanism to handle groups of operations as though they were one.
- Either all operations in a transaction occur or none occur at all.
- The operations involved in a transaction might rely on one or more databases.

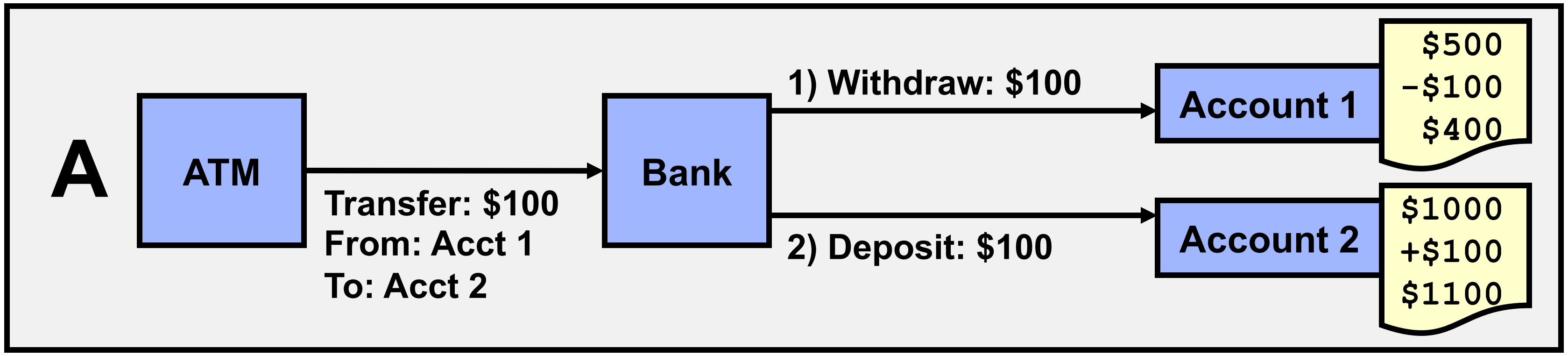
ACID Properties of a Transaction

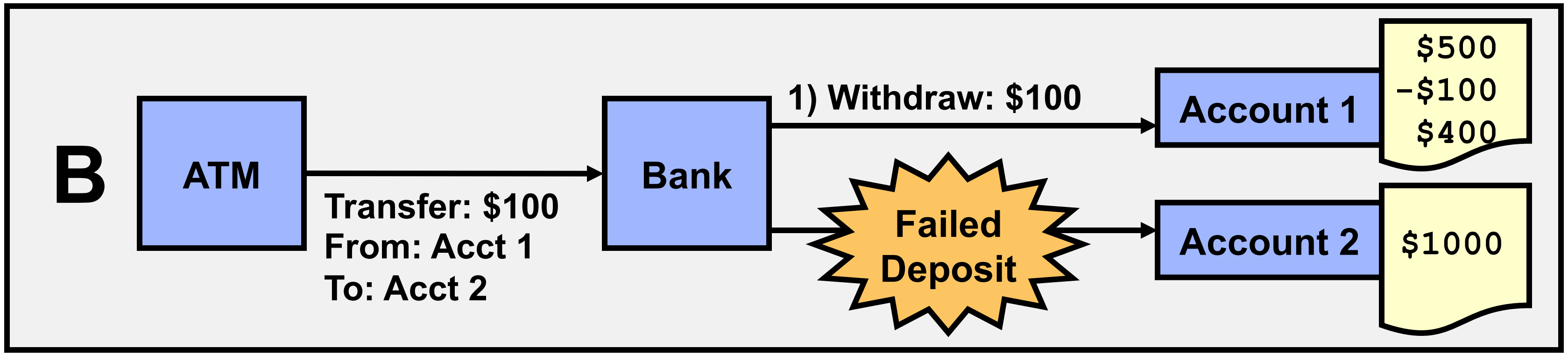
A transaction is formally defined by the set of properties that is known by the acronym ACID.

- **Atomicity:** A transaction is done or undone completely. In the event of a failure, all operations and procedures are undone, and all data rolls back to its previous state.
- **Consistency:** A transaction transforms a system from one consistent state to another consistent state.
- **Isolation:** Each transaction occurs independently of other transactions that occur at the same time.
- **Durability:** Completed transactions remain permanent, even during system failure.

Transferring Without Transactions

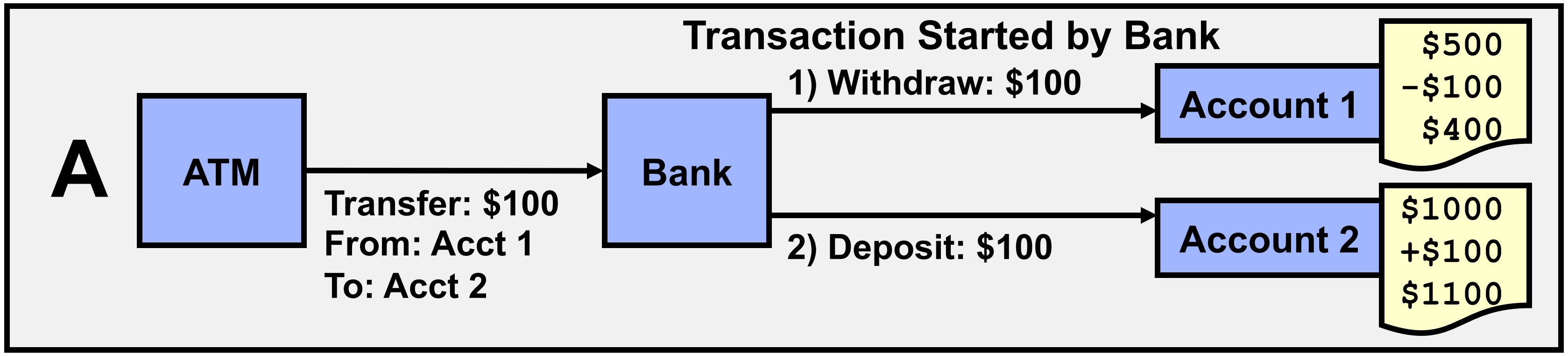
- Successful transfer (A)
- Unsuccessful transfer (Accounts are left in an inconsistent state.) (B)

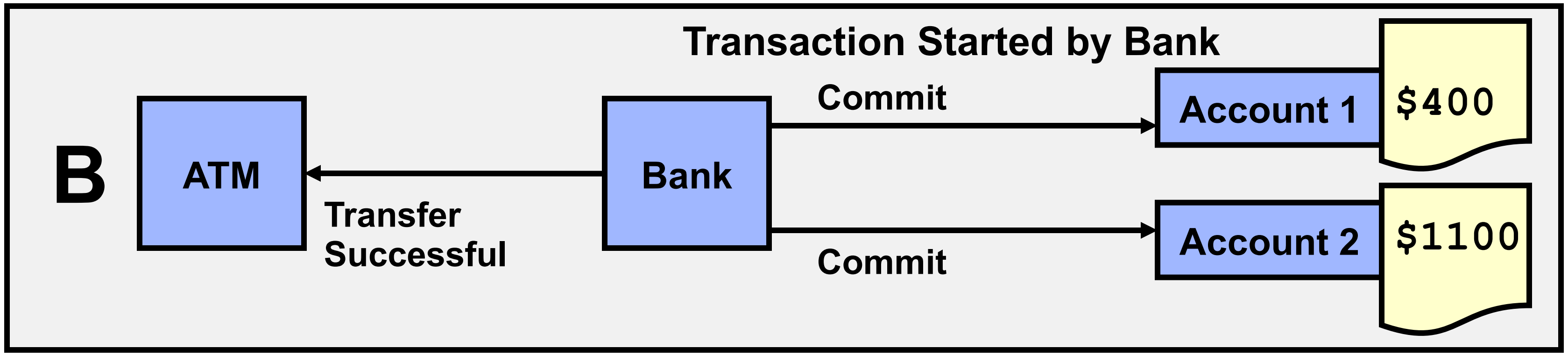




Successful Transfer with Transactions

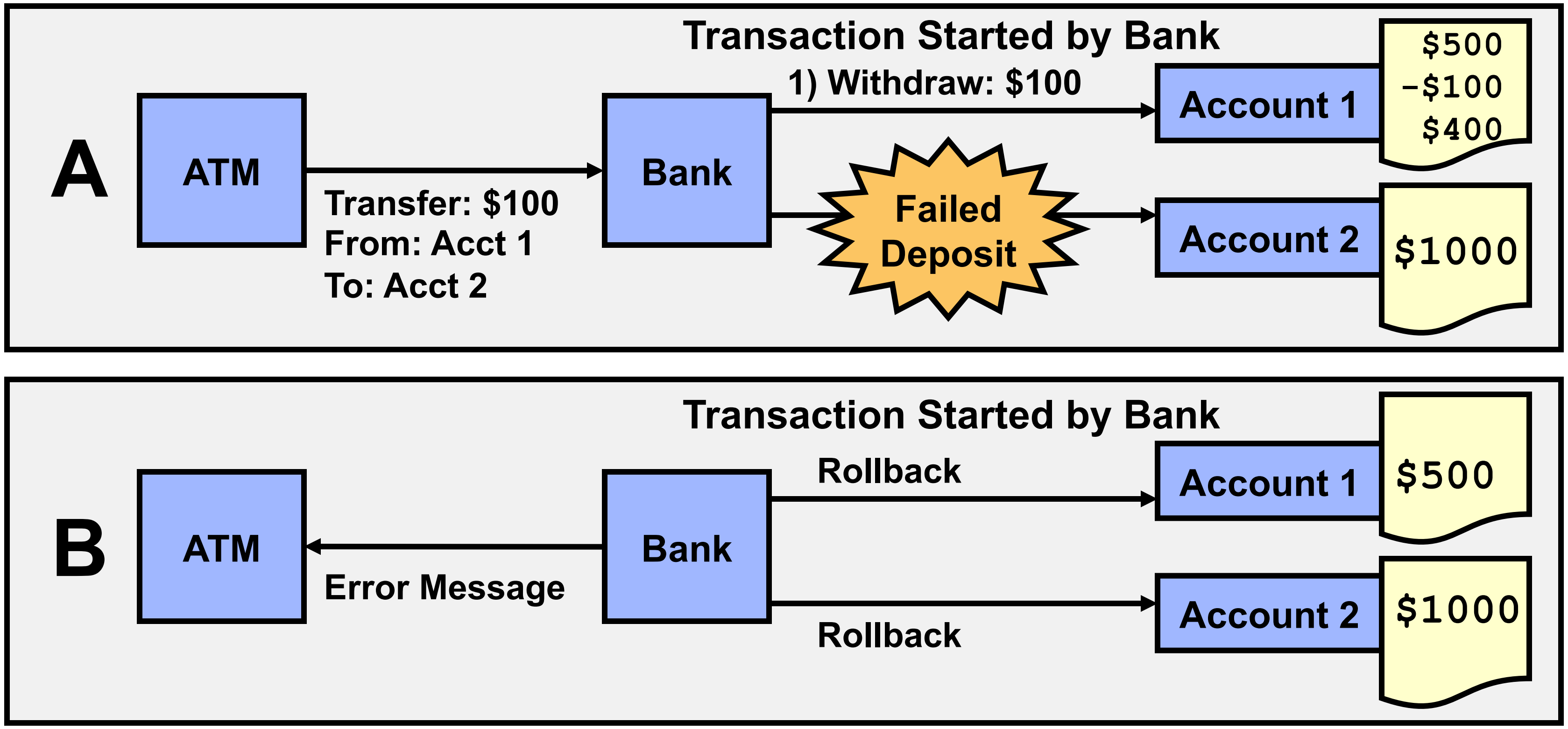
- Changes within a transaction are buffered. (A)
- If a transfer is successful, changes are committed (made permanent). (B)





Unsuccessful Transfer with Transactions

- Changes within a transaction are buffered. (A)
- If a problem occurs, the transaction is rolled back to the previous consistent state. (B)



JDBC Transactions

By default, when a `Connection` is created, it is in auto-commit mode.

- Each individual SQL statement is treated as a transaction and automatically committed after it is executed.
- To group two or more statements together, you must disable auto-commit mode.

```
con.setAutoCommit (false);
```

- You must explicitly call the commit method to complete the transaction with the database.

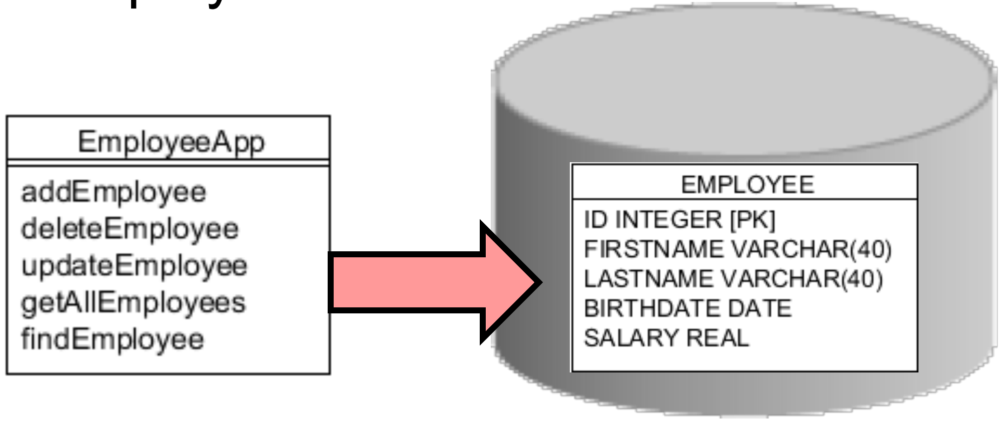
```
con.commit();
```

- You can also programmatically roll back transactions in the event of a failure.

```
con.rollback();
```

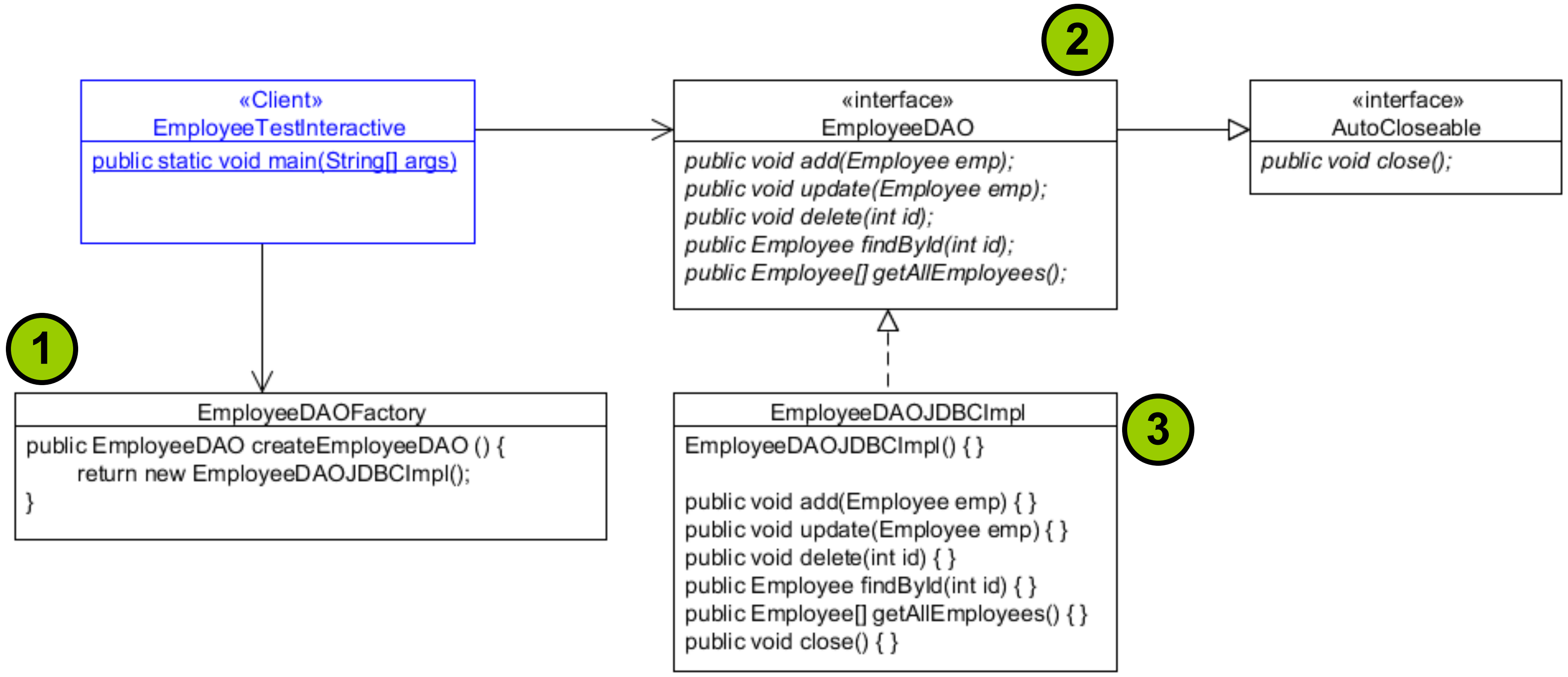
Data Access Objects

Consider an employee table like the one in the sample JDBC code.



- By combining the code that accesses the database with the "business" logic, the data access methods and the Employee table are tightly coupled.
- Any changes to the table (such as adding a field) will require a complete change to the application.
- Employee data is not encapsulated within the example application.

The Data Access Object Pattern



«Client»
EmployeeTestInteractive
public static void main(String[] args)
«interface»
EmployeeDAO
public void add(Employee emp);
public void update(Employee emp);
public void delete(int id);
public Employee findById(int id);
public Employee[] getAllEmployees();
«interface»
AutoCloseable
public void close();
1
EmployeeDAOFactory
public EmployeeDAO createEmployeeDAO () {
return new EmployeeDAOJDBCImpl();
}
2
3
EmployeeDAOJDBCImpl
EmployeeDAOJDBCImpl() { }
public void add(Employee emp) { }
public void update(Employee emp) { }
public void delete(int id) { }
public Employee findById(int id) { }
public Employee[] getAllEmployees() { }
public void close() { }