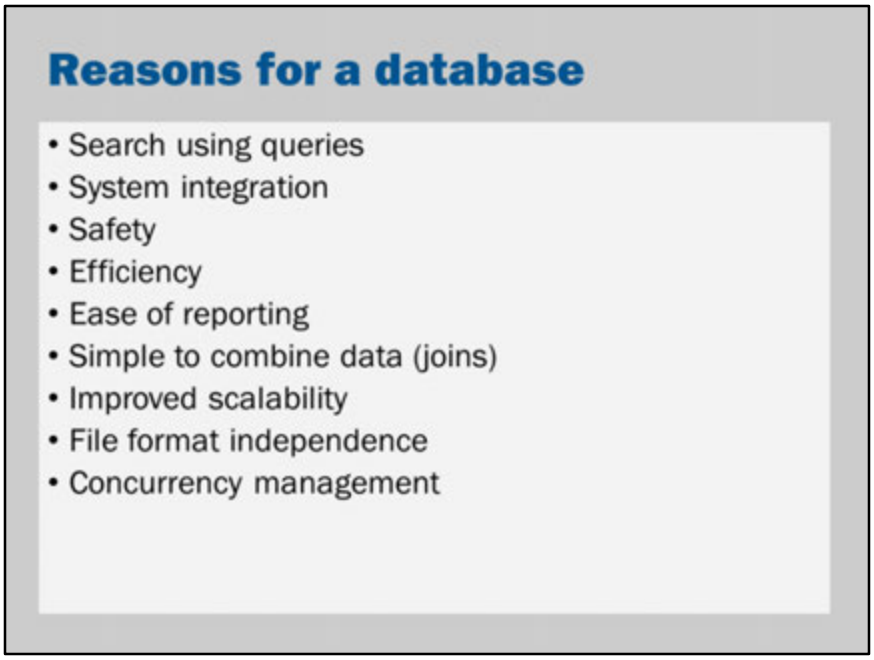
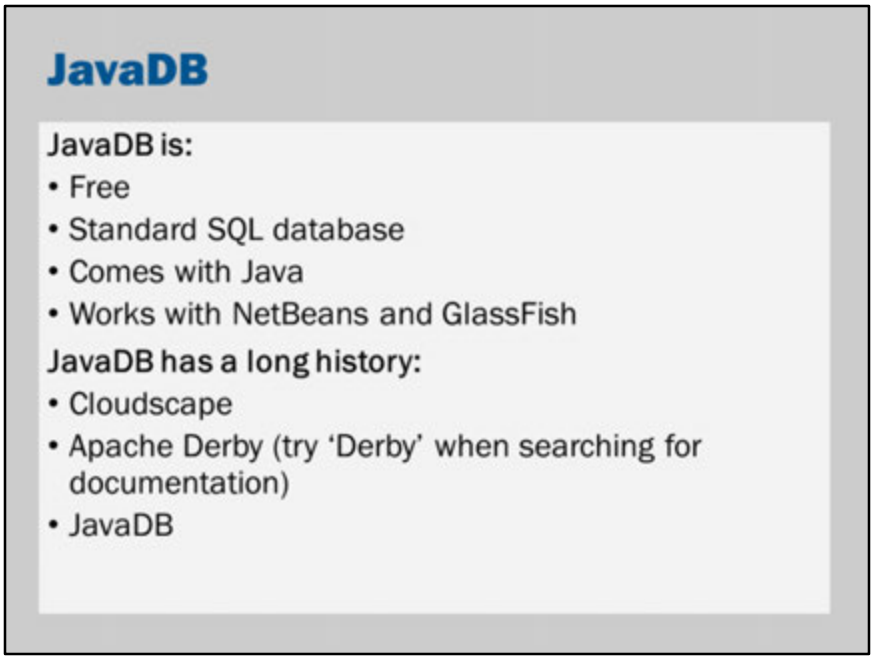
Why databases?



Why use a database?

- You can query the data (run searches)
- You can integrate with other business systems that use the same database
- You can store huge numbers of records without the risk of corruption
- You can access data quickly because the database creates efficient “indexes”
- You can generate reports quickly
- You can combine multiple types of data (using table joins), without needing to implement complex code
- You can take advantage of database features that provide reliability, recoverability, fault tolerance, replication and scalability
- You do not have to worry about file formats when storing data
- You do not have to worry about concurrent users: the database ensures that changes do not interfere with each other



JavaDB is a free database. It is installed with Java.

We are using it in this subject because:
- It is free
- It doesn't require installation (you can use it on your laptop)
- It is pure Java
- It works well with NetBeans/GlassFish
- It supports everything you need for this subject

You are welcome to experiment with other databases.
The major database vendors provide free developer/student/express downloads of their databases:
- Oracle Database
  http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html
- IBM DB2
  http://www-01.ibm.com/software/data/db2/express-c/download.html
- Microsoft SQL Server
  https://www.dreamspark.com/Student/Software-Catalog.aspx

JavaDB has a long history with the Java platform:

- First created by Cloudscape, Inc as a pure Java database in 1997
- Informix bought Cloudscape (1999)
- IBM bought Informix and IBM Cloudscape was possibly the most popular pure Java database (2001)
- IBM donated the code to Apache; the system became known as Apache Derby (2004)
- Oracle now includes Apache Derby with Java. They call it JavaDB.

This long history means that if you have a question about JavaDB, you may find more results if you search for Derby instead.

In this demo, I show how to configure JavaDB.
I create a new database named "aip", with username "aip" and password "aip".
I ask that you use this configuration for your assignment to assist with marking.
Of course, you are free to use other databases for your assignment.

I also show how to execute SQL queries from within NetBeans.

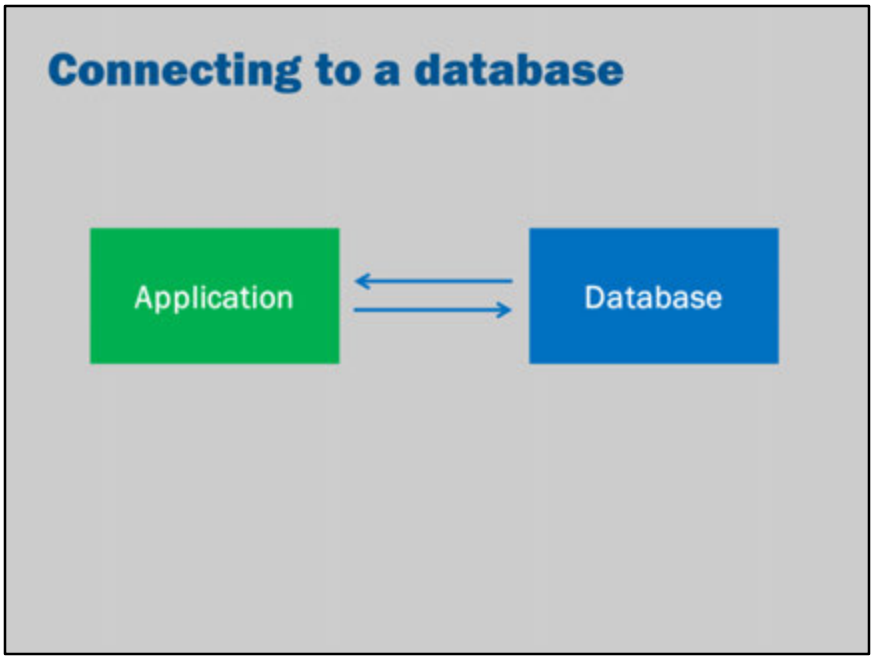
JDBC

Most SQL databases work in the same way:

- Send the database an SQL query
- The database responds with tabular data

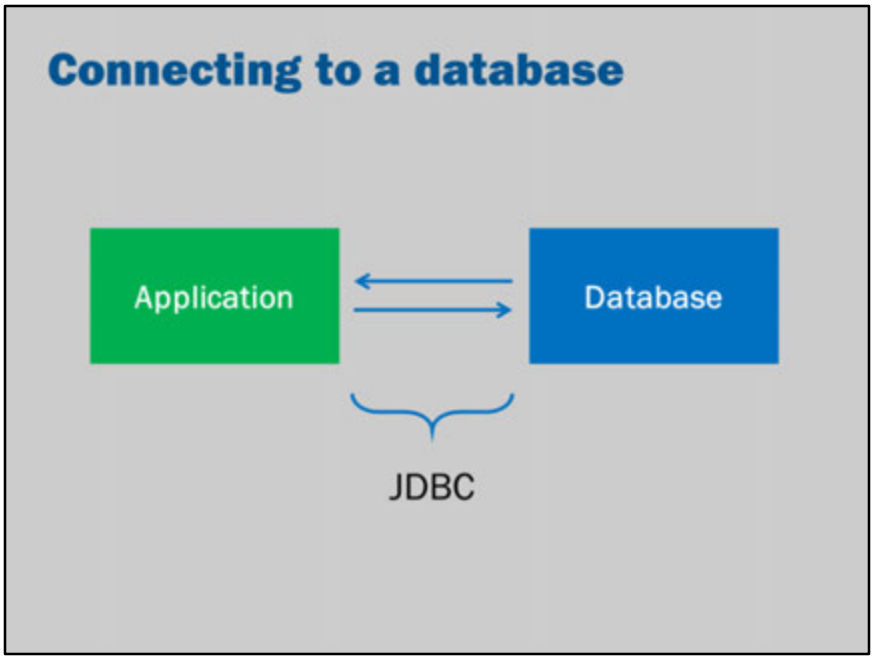
JDBC provides a standard interface

JDBC



The objective of this video is to explore:

- How we can connect our application to a database
- How we can design our applications that connect to a database



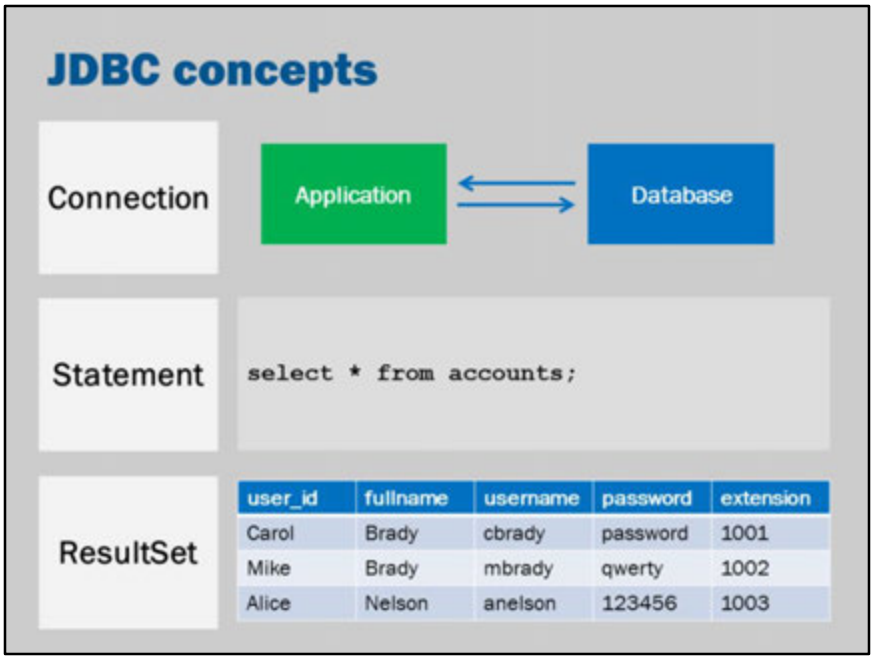
JDBC is the a standard Java API for talking to databases.

JDBC is not officially an acronym (it is trademarked as "JDBC").
However, it is often read as “Java Data Base Connectivity”

Java usually uses standardization to introduce new technologies.
A standards process defines a common API (i.e., interfaces and classes that need to be implemented).
Third party vendors then implement the API.
This means that database vendors typically offer their own implementation of JDBC:

- Oracle provides a JDBC driver
- Microsoft provides a JDBC driver for SQL Server
- IBM provides a JDBC driver for DB2
- and, of course, JavaDB (Derby) comes with its own JDBC driver.

Once you learn JDBC, you can use that knowledge with any database.
If you write a program using JDBC, that program can (in theory) be made to use a different database.



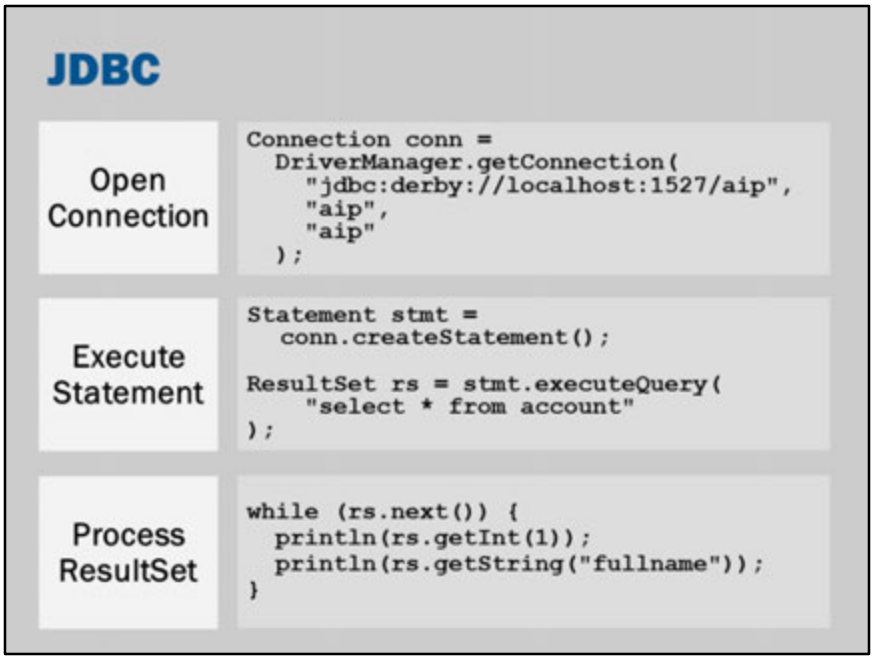
There are three important concepts in JDBC:

Connection
A connection is a link between a Java program and a database.
Often this corresponds to a network connection (i.e., a TCP/IP connection).
The connection is used to send SQL commands to the database and to return the results.

Statement
A statement is a single SQL query or command.
A statement is associated with a connection.
The statement can be executed on that connection.
When it is executed, the database will return the result from the query (i.e., a ResultSet).

ResultSet
A result set is data in a table that comes back as a result of executing a statement.

We might say:
A connection is used to produce/execute statements.
Statements are used to produce results.
i.e., Connection -> Statement -> ResultSet



Connection

To create a connection, we use a connection string. The connection string tells JDBC and the JDBC driver which database to access and how to access it.

In this example, the connection string tells us:

- **jdbc:** the name of the protocol
- **derby:** the name of the database driver (i.e., Java DB)
- **//localhost:1527/** the address and port of the database server
- **aip** the name of the database file or catalog to use

In addition, the username and password for the database connection is supplied (username: aip, password: aip).

Statement

When we create a statement, it is associated with a connection.

A statement, once created, can be used to execute a query, perform an update, or call a stored procedure.

We pass a string containing SQL to the statement.

The statement uses the connection to perform the query and returns a ResultSet

ResultSet

A ResultSet is a row-based table.
You step over the table row-by-row.

To move to the first row, you would call the next() function.
It returns true if the row is available.
Calling next() again would move to the next row.
It returns true, once again, if the row is available.
This means that you can iterate over the entire table by calling:

```
while (rs.next()) {
  // process a row
}
```

When you're on each row, you can retrieve the values of columns.
If the first column is named "user_id", then you can access the first row in two ways:

1. rs.getInt(1) retrieves the value of the first column of the current row.
2. rs.getInt("user_id") retrieves the value of the column named "user_id" of the current row.

ResultSet has various functions to retrieve the values of columns according to the type of the column:
For example,

- rs.getInt(x)
- rs.getString(x)
- rs.getDate(x)

You need to know the type of the column.
However, conversion between (compatible) Java types and SQL types is automatic.
If you try to call rs.getInt(x) on a column that is a VARCHAR, you will have an error.
If you call rs.getString(x) on a column that is an INTEGER, then Java will convert the number to a string.

Close everything

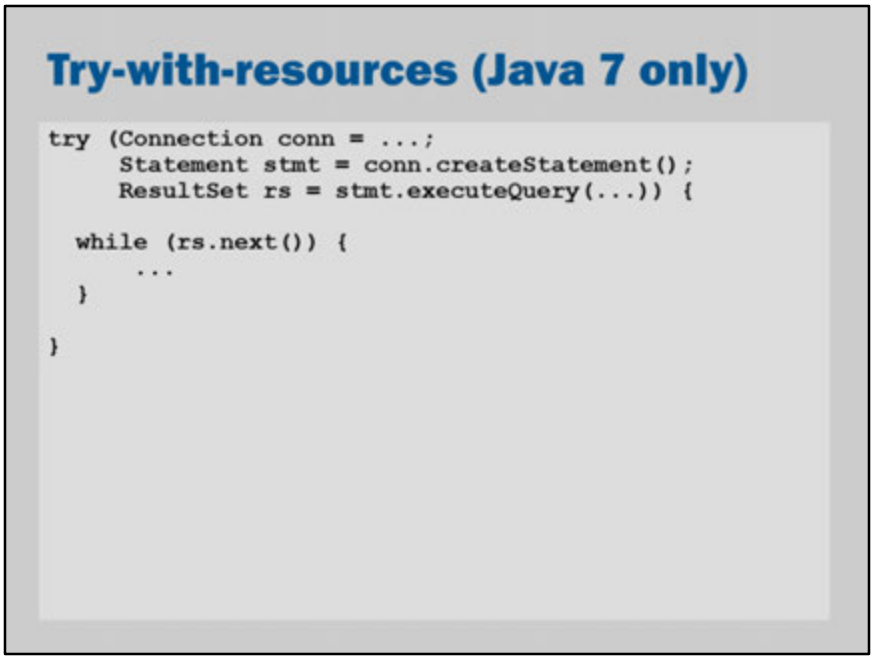
```
Connection conn = DriverManager.getConnection(...);
try {
  Statement stmt = conn.createStatement();
  try {
    ResultSet rs = stmt.executeQuery(...);
    try {

      while (rs.next()) {
        ...
      }

    } finally {
      rs.close();
    }
  } finally {
    stmt.close();
  }
} finally {
  conn.close();
}
```

It is important to close everything you open or create.
This is to ensure that connections don't stay open.
This is to ensure that resources aren't wasted when you're finished your query.
There are memory and connection limits (due to your operating system and also due to physical limits).
Therefore, keeping connections open can mean that eventually your web site "runs out" and stops working.

According to the specification, closing the connection closes everything else.
So, in theory, this is all you need to do.
However, it is good practice to close everything:

- The driver may be have bugs
- Your code may have bugs
- Closing makes it clear in your code when you're finished with the resource

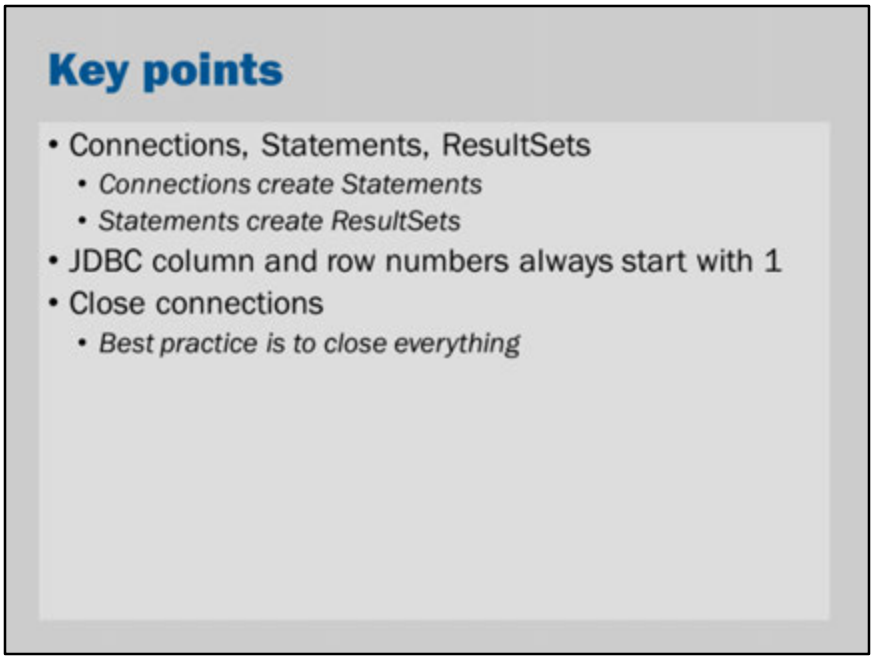
If you are doing more than one thing on the connection, then it certainly makes sense to close your ResultSets and Statements.
This frees up database resources (such as the "cursor" used by the database to move through the results, and also any locks that may have been used on the database).



A new feature of Java 7 is the try-with-resources syntax.
The try-with-resources syntax provides a convenient way to automatically close resources that are created/opened.

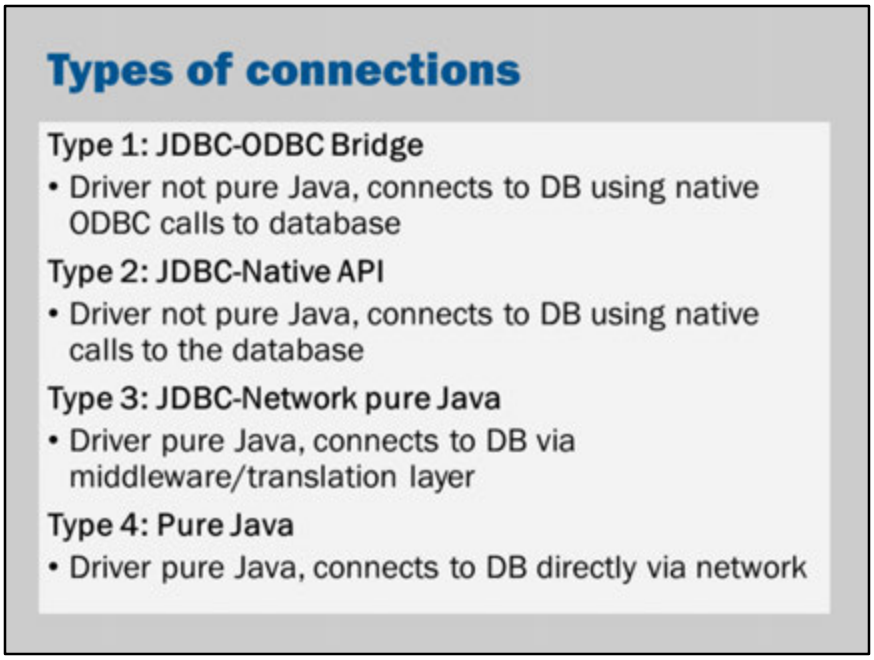
try (GET-RESOURCES) {
  CODE-THAT-USES-THE-RESOURCES
}

Each statement inside the brackets that open a try-with-resources is automatically closed at the end of the try statement.
They are closed, even if the code inside the try statement throws an exception or includes a return statement.



Note also: In theory, JDBC is thread safe but it is better to keep one connection per thread

JDBC performance

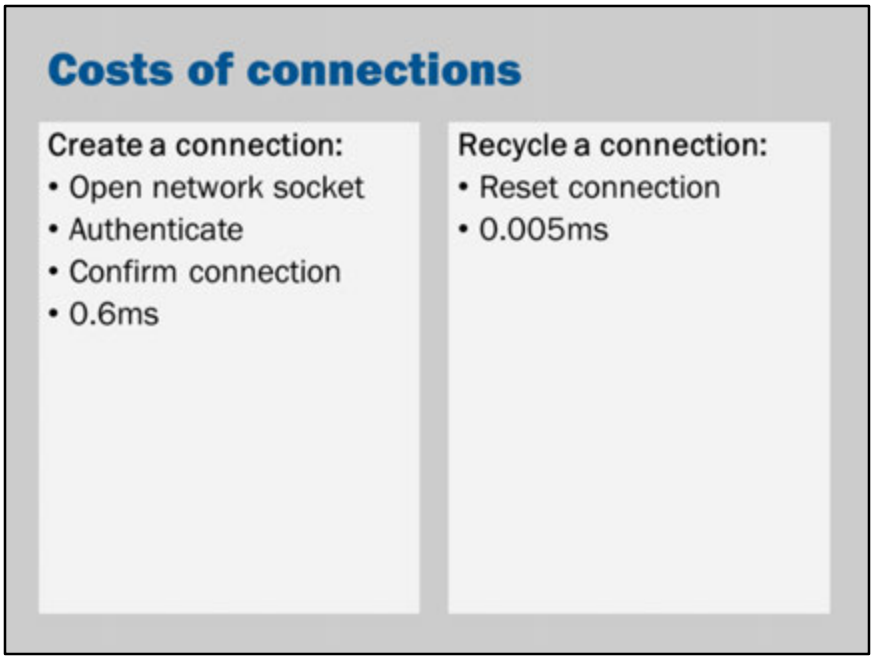


JDBC drivers provide the vendor-specific code for connecting to a database.
That is, the driver is what provides the facility of connecting to a specific database.

JDBC drivers have been designed in different ways.
Type 4 is generally preferred by Java developers.
This is because it tends to be more portable (i.e., use on Unix, Windows and Mac), faster, less resource intensive, more reliable and requires less installation.
However, sometimes it isn't always possible to get a Type 4 driver.

Typically, if you can't get a type 4 driver, you'll fall back to the JDBC-ODBC bridge.
ODBC is a cross-platform database connectivity API designed for C code.

Read more: http://en.wikipedia.org/wiki/JDBC_driver



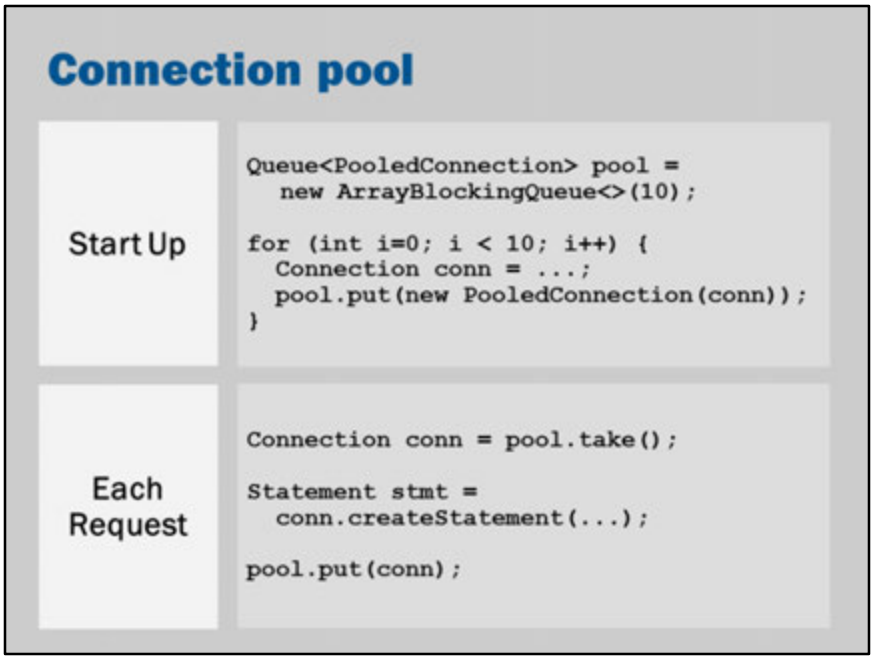
Even with a Type 4 driver, opening a connection can be expensive.

I ran a test on my computer that opened thousands of connections.
On average, it takes approximately 0.6 milliseconds to open and close a connection.

This sounds like a small number.
However, for your computer it is a very long time.
If your website gets 1700 requests per second, the computer will be spending that entire second just opening and closing connection.

On the other hand, it is possible to intercept the "close" and then "recycle" an existing connection.
By reusing the connection, there is no need to use all those time/resources involved in establishing the connection.

On my computer, a connection pool was able to serve a connection in 0.005 milliseconds.
The same 1700 requests would take only 8.5 milliseconds to handle connection open/closing.



A connection pool is a service provided by an application server.
It keeps a number of connections in memory, ready to be given on request.

The code you see is not real code.
It is just "example code" or "pseudocode" that illustrates how a connection pool works.

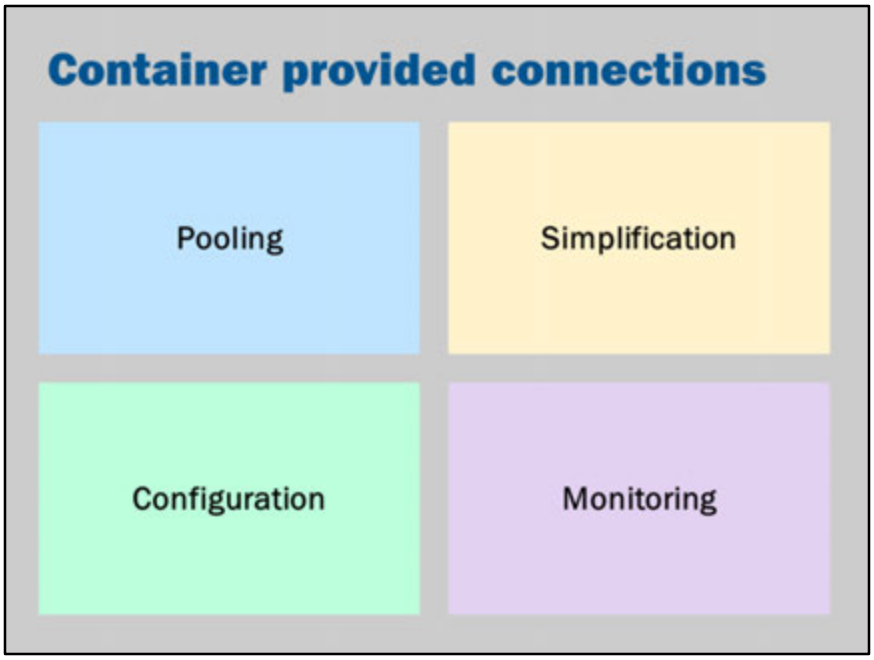
The application server hides all these details behind the scenes.

Using a data source

```
@Named
@RequestScoped
public class MyController {

public void query() throws SQLException,
  NamingException {

    DataSource ds = InitialContext.doLookup(
      "jdbc/aip"
    );
    String query = "select * from account";
    try (Connection conn = ds.getConnection();
         Statement stmt = conn.createStatement();
         ResultSet rs = stmt.executeQuery()) {
      while (rs.next()) {
        ...
      }
    }

  }
}
```

To create a connection from within a Java EE application server or web server:

1. Get a DataSource via a lookup (Technically speaking, this is a JNDI lookup. We will cover JNDI in detail in a future lecture.)
2. Use the DataSource to get a connection from the connection pool
3. Use the connection
4. Close the connection (the "close" is intercepted so it doesn't actually close the underlying connection)



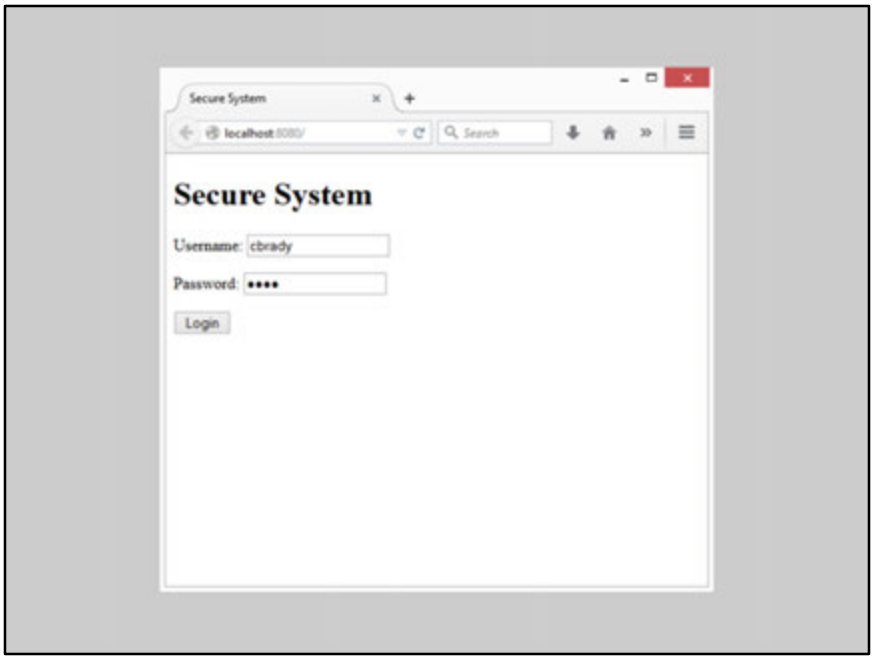
Connection pooling is one service provided by the container.
Using container managed connections offers other benefits:

- **Configuration**
  The container can remember the connection string, database username and password.
  In addition, the JDBC driver may have other advanced configuration options that can be centralized in the application container.
- **Simplification**
  Using the container for connections can ensure that connection establishment and configuration code is in one place.
  Not only that, but the connection may be configured and changed in a GUI or configuration file.
- **Monitoring**
  When the application server provides connections, it can also monitor for issues with the database.
  Bad connections, faulty connections and other issues can be detected by the application server.

Key points

- Request JDBC connections from the application server (via a DataSource)
- Remember to close everything!

JDBC details



For this example, imagine a simple login system.

Login code (with problems)

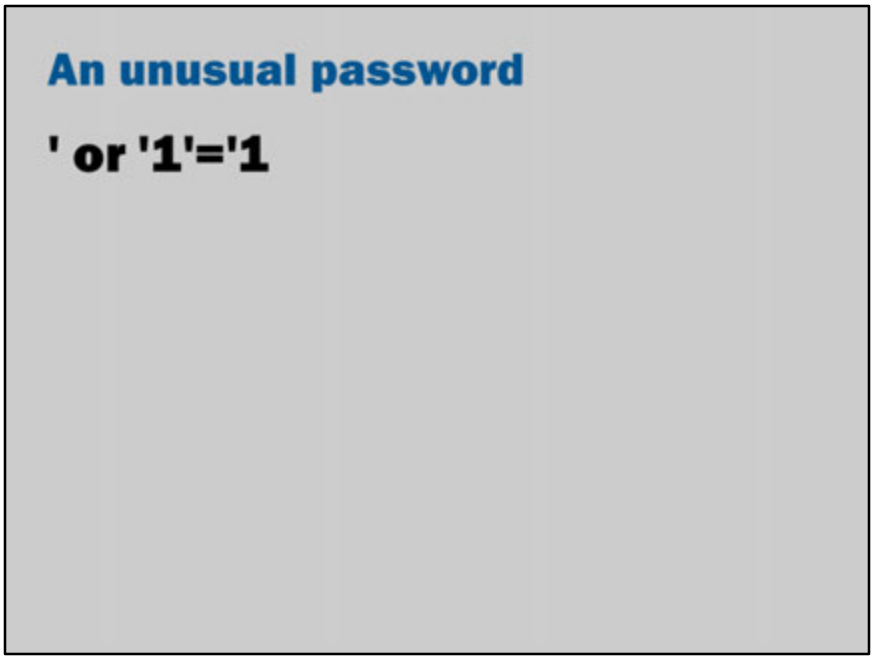
```
private String checkLogin(String username,
                          String password)
                  throws SQLException, NamingException {

  DataSource ds = InitialContext.doLookup("jdbc/aip");

  try (Connection conn = ds.getConnection();
     Statement stmt = conn.createStatement()) {

    String query =
      "select user_id from account " +
      "where username = '" + username + "' " +
      "   and password = '" + password + "' ";

    try (ResultSet rs = stmt.executeQuery(query)) {

      if (rs.next()) {
        return rs.getString("user_id");
      }

    }
  }
  // no user found
  return null;
}
```

Here's some code that might handle the login.
It takes a username and password, and then executes an SQL query.
It creates the SQL query by joining together some strings.

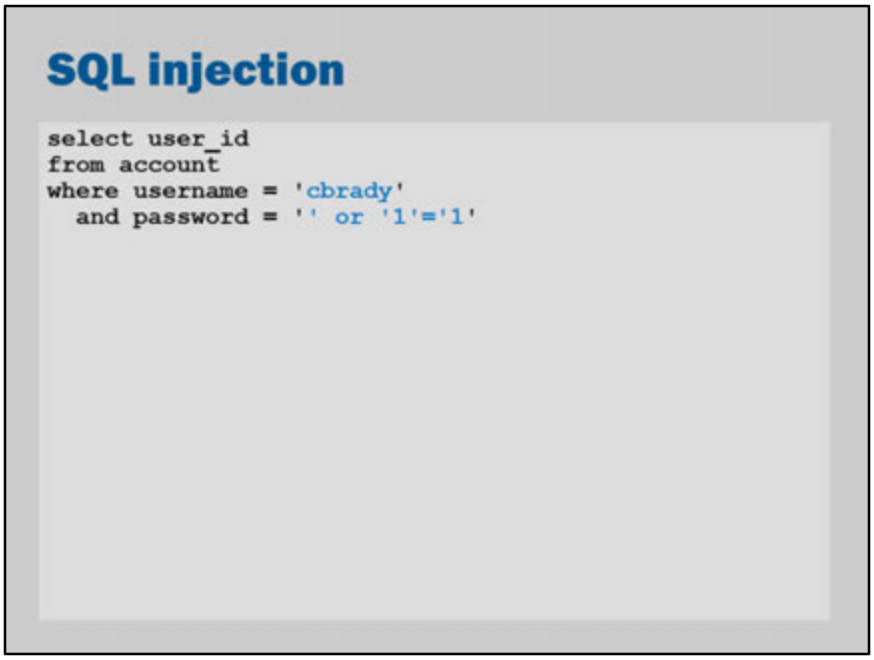
Using parameters in queries

```
select user_id
from account
where username = 'cbrady'
  and password = 'asdf'
```

This is an example of the query that was sent to the database when we logged in. The query is used to check that the user's login credentials are correct.



But what happens if, instead of entering the password "asdf", the user enters a password such as "' or '1'='1"?



A danger of simply combining text strings is that the user could enter inputs that change the SQL query.

In the example above, the user has entered a password that contains quotes: ' or ''='

The SQL query does not properly check that the password matches.
The password='' will fail. However, the other condition in the *or*-expression will succeed because the string containing 1 ('1') will always be equal to the string containing 1 (i.e., '1'='1' is always true).
This means that the system will successfully log in, regardless of the user's real password.
A malicious hacker can use this to gain access to accounts.

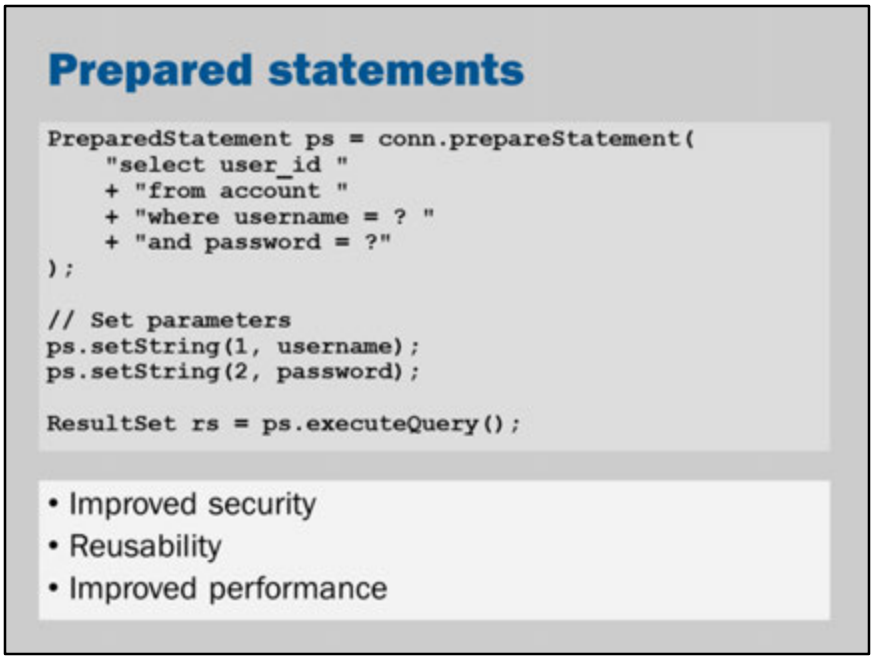
This kind of attack is well known. It is called "SQL injection".
https://en.wikipedia.org/wiki/SQL_injection

SQL injection

```
select user_id
from account
where username = 'cbrady'
  and password = '' or '1'='1';
drop table account;
--'
```

There are dangers in addition to hackers gaining unwanted access.
In theory, with a carefully crafted query, they could do even more damage.

In JDBC, this particularly query will not work.
Statement.executeQuery normally only allows a single query.
Nevertheless, this kind of "SQL injection" is something to be very cautious about.



In theory, we could write code to escape the user input (e.g., replacing quotes and other special characters with "safe" input).

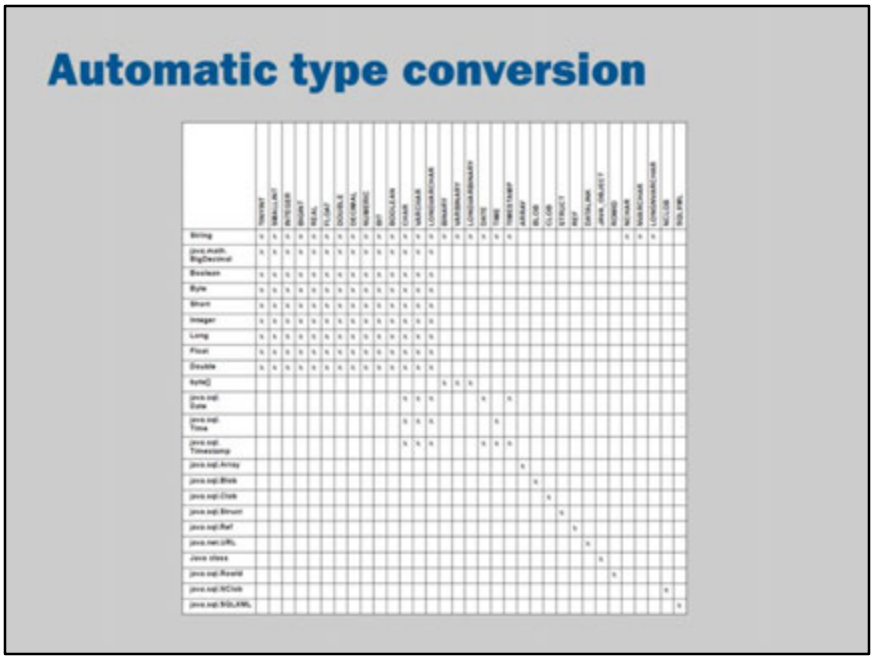
A better approach is to use the PreparedStatement feature of JDBC.

A prepared statement is a query with placeholders identified by question marks.
The values of the question marks can be set before executing the query.
The database driver will replace the question marks with the values that have been set.

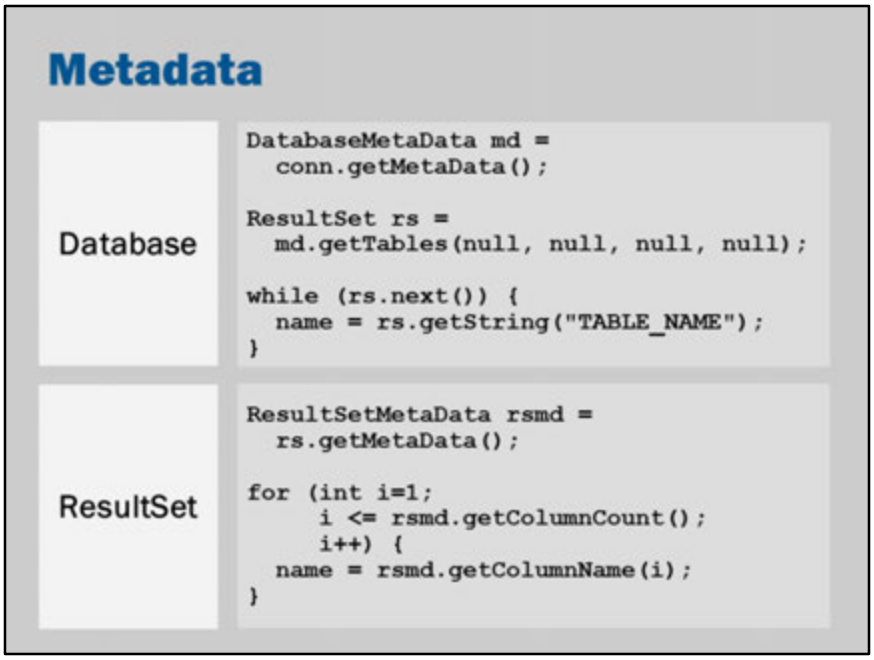
In addition to security, PreparedStatement are "compiled" once by the webserver and can be used multiple times.
Reusing the prepared statement improves performance.
This is because the query does not need to be compiled with every request.

Benefits aren't just security. A stored procedure can be created once and reused many times.

JDBC has a number of other features that are helpful.



JDBC automatically converts between Java types and SQL data types.
This matrix illustrates the many possible automatic conversions.
Note that most (but not all) types can be converted into Strings.



JDBC has an ability to retrieve metadata.
In other words, you can get the database schema from the database.
This may not be helpful when you know the database schema.
However, it can be used to create tools that explore the database or allow user-entered queries.

Metadata can be used to get a list of all tables, all views and capabilities of the database.

For a full list of schemas and columns, refer to the JDBC specification.

Insertion / update

```
try (Connection conn = ds.getConnection();
     Statement stmt = conn.createStatement()) {
  stmt.execute("insert into accounts " +
               "(username, password) values " +
               "('mbrady', 'qwerty')"

  int rowsUpdated = stmt.executeUpdate();

  if (1 != rowsUpdated) {
    // handle update problem
  }
}
```

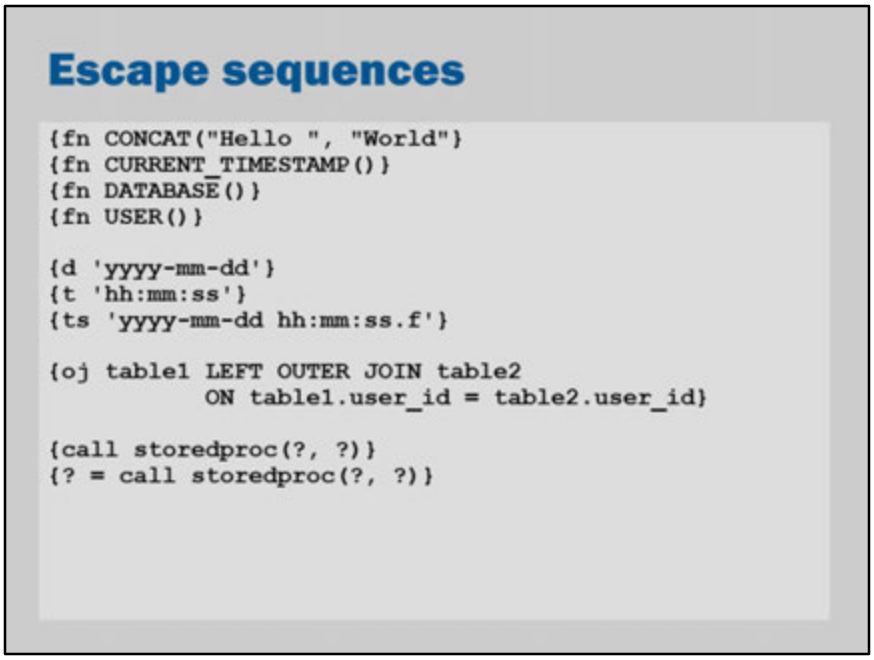
In addition to queries, JDBC supports updates.
You can execute, insert and delete statements using a statement

ResultSet updates

```
try (Connection conn = ds.getConnection();
     Statement stmt = conn.createStatement(
       ResultSet.TYPE_SCROLL_SENSITIVE,
       ResultSet.CONCUR_UPDATABLE);
     ResultSet rs = stmt.executeQuery(query)) {

  if (rs.next()) {
    rs.updateString("fullname", "Carol Martin");
    rs.updateRow();
  }
}
```

Java allows you to modify a ResultSet.
This is in addition to being able to run update queries directly as we saw on the previous slide.

Set the ResultSet.CONCUR_UPDATABLE option and then call update methods on the rows.
To confirm the row changes to the database, you need to call updateRow.



You can always use the SQL syntax specific to your database.

However, JDBC provides standard escape sequences.
The sequences are a cross-platform, database vendor independent syntax.
They allow you perform enhanced SQL tasks in a way that works on multiple databases.

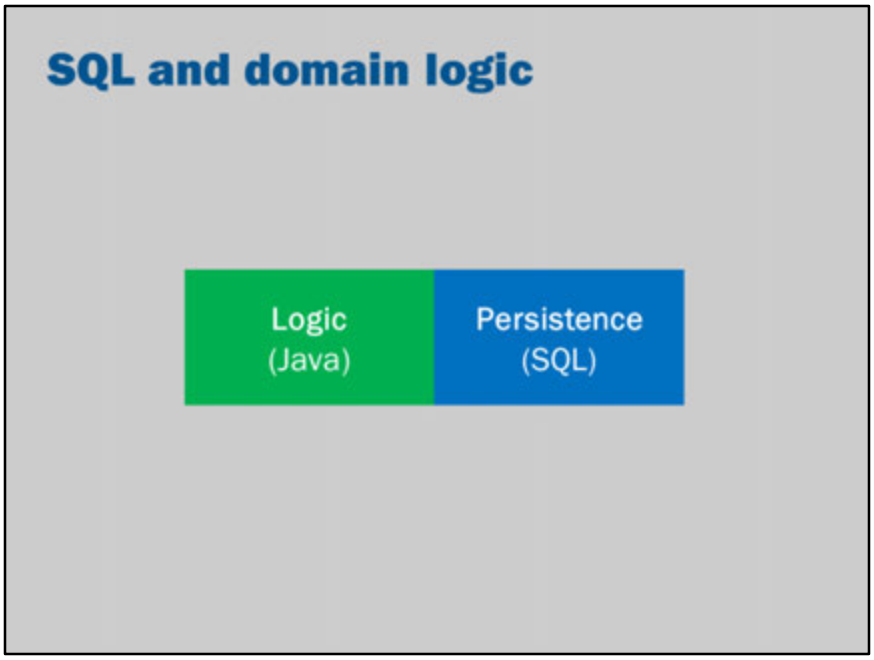
Refer to Section 13.4 and Appendix C of the JDBC specification:
https://jcp.org/aboutJava/communityprocess/final/jsr221/

Key point

- Always use a PreparedStatement when processing user input

Once again, good design requires us to separate layers and code as much as possible. Database access code and SQL code are specific to the storage or persistence mechanism.

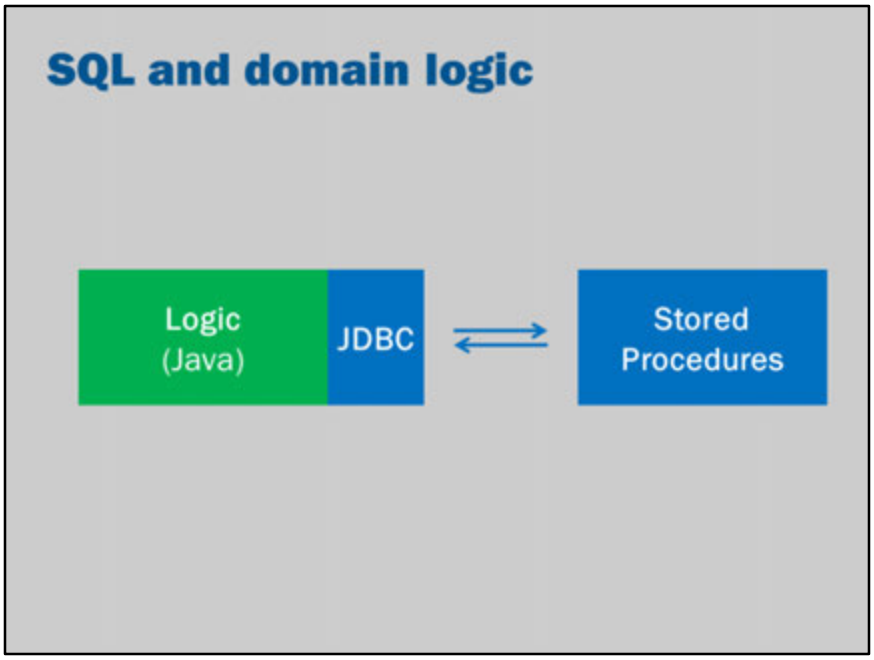
What happens if we want to replace our SQL database with some other storage technology?
e.g., object database
e.g., file system
e.g., in-memory only
e.g., storage in legacy system
e.g., storage on remote system



One approach to web development is just to put everything together in one big mess.

This creates the problem of having domain logic and persistence (JDBC) code in the one place.

What can we do to improve the design?



A stored procedures is code that runs inside the database.

On Oracle, DB2, Microsoft SQL Server and other databases, stored procedures are written using an extended dialect of SQL.

- Oracle uses PL/SQL
- DB2 uses SQL Procedural Language
- Microsoft SQL Server and Sybase products use Transact-SQL

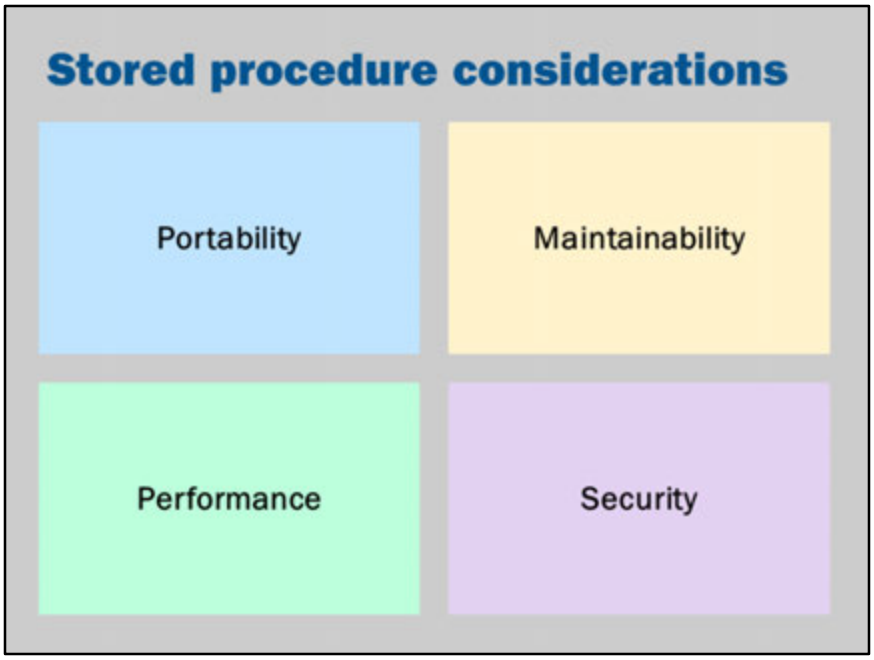
In JavaDB/Derby, stored procedures are written in Java. Java classes are loaded into the database and run inside the database.

Essentially, Stored Procedures are one way of moving some of the persistence logic into the database.

Many large organizations have multiple applications running on one database. Some organizations use stored procedures to implement application-independent domain logic.

Stored procedures aren't a great solution to separating concerns into layers.

This is because Java JDBC code is still required to call the stored procedure.
We still have the problem of separating the JDBC code from the domain logic in the application.



This question relates to an old debate in database design:
Should we use stored procedures or should we keep logic in our application.

Database Administrators (DBAs) often argue for stored procedures.
Programmers often argue for keeping the SQL queries and logic in the application programming language.

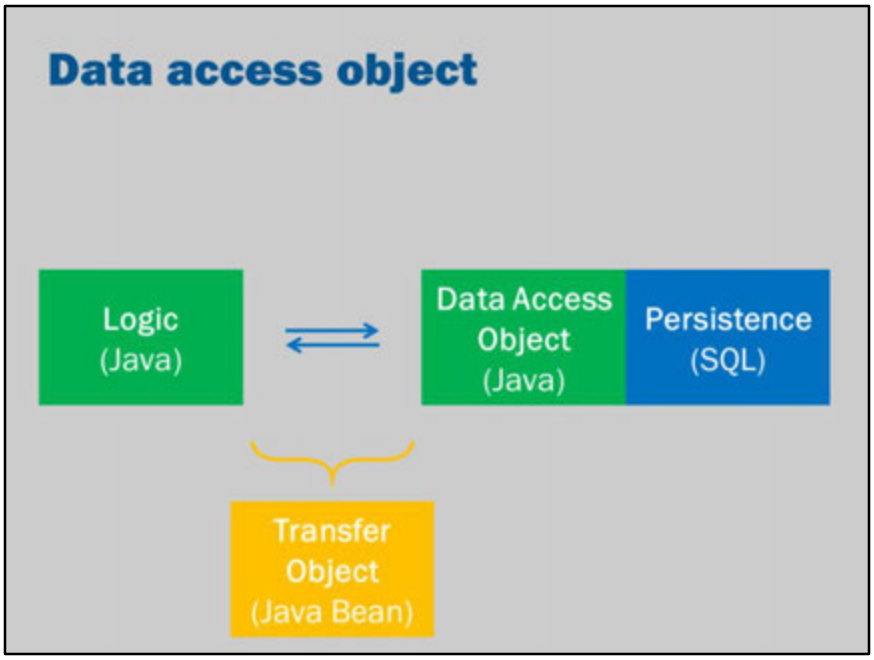
This is a very old debate.
https://www.simple-talk.com/sql/t-sql-programming/to-sp-or-not-to-sp-in-sql-server/
http://web.archive.org/web/20110814035521/http://www.theserverside.net/news/thread.tss?thread_id=31953

Some considerations:

- **Portability**
  Java is platform independent and database independent. With Java code, you can switch databases.
  In contrast, PL/SQL must always run on an Oracle database.
- **Performance**

Stored procedures are often slightly faster. The database can precompile the queries and spend more time identifying an efficient plan.
In addition, stored procedures may be easier for a DBA to optimize.

- **Maintainability**
  Stored procedures may not be part of the application source code, so can be difficult to maintain.
  In addition, the database may have different release cycles than the application.
- **Security**
  Stored procedures can enforce consistent security policies throughout an entire application and throughout an entire organization.

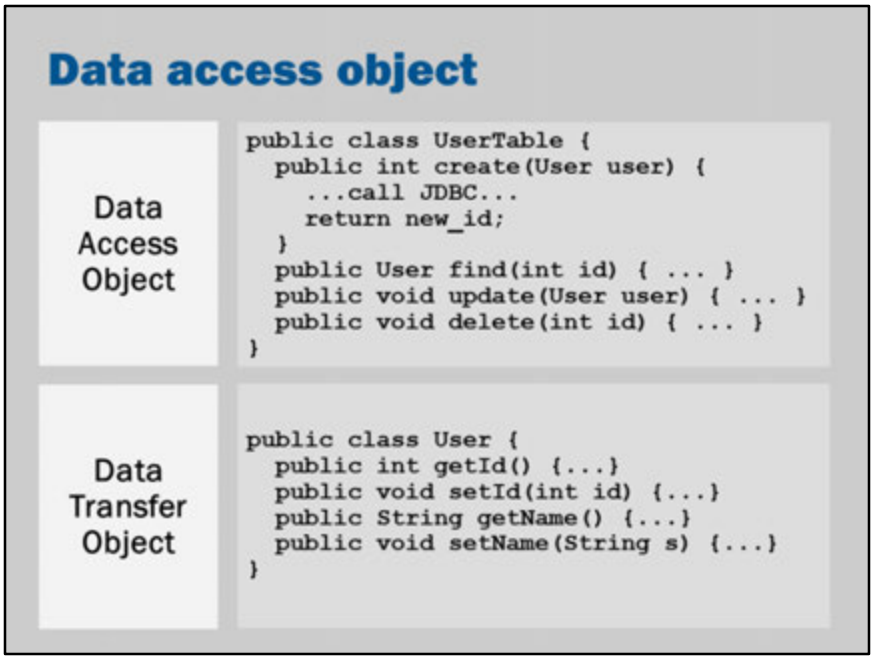


An alternative design is to separate code that accesses the database into its own layer.
Put the domain logic in another layer.
Then have the two layers communicate with each other using data transfer objecs.

A Data Access Object (DAO) is an object (i.e., a Java class) that is solely responsible for storing or retrieving data in a database.
Data is given to the DAO by a object known as a Data Transfer Object (DTO).

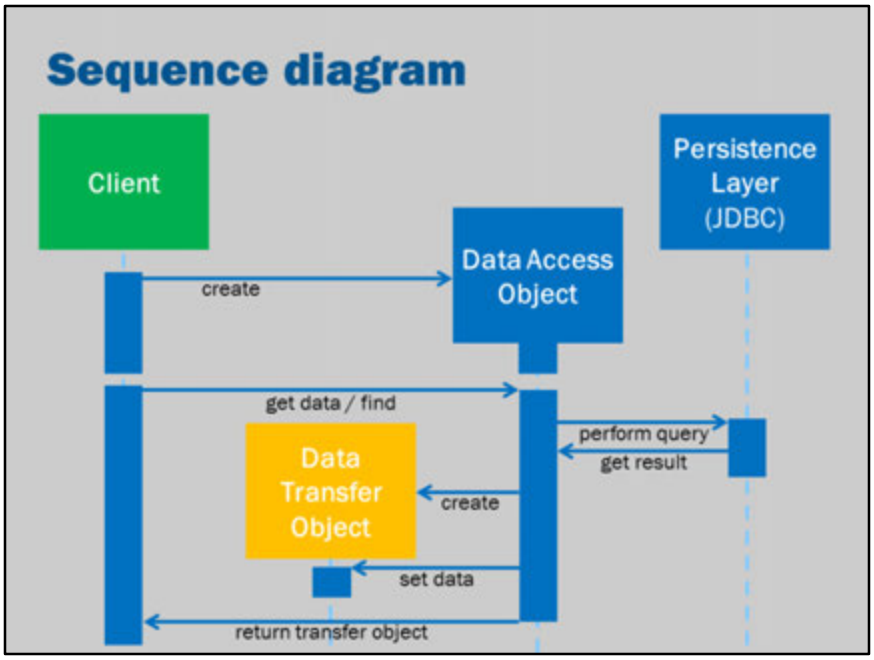
A DTO is typically a Java bean that just holds the data in a row.

For more detailed information:
http://www.oracle.com/technetwork/java/dataaccessobject-138824.html
http://tutorials.jenkov.com/java-persistence/dao-design-pattern.html



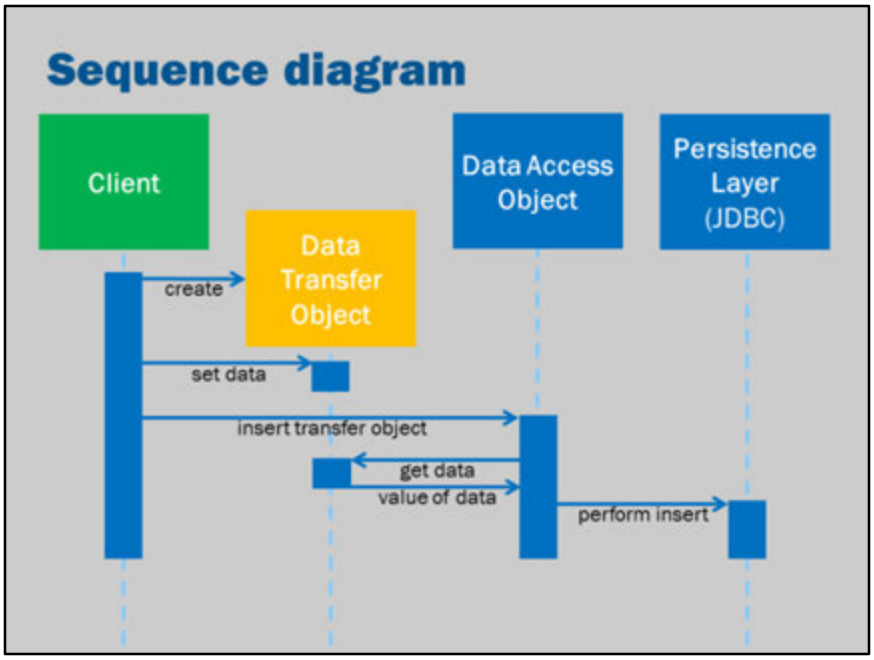
This shows the basic structure of a DAO and DTO.
Note that the DTO is just a simple "container" that stores the columns of a record.
It does not include any domain logic.

The DAO contains only the logic for storing (and retrieving) the data in a DTO into the database.



This is a UML sequence diagram that illustrates the flow of control (for a read/find/get function on the DAO):

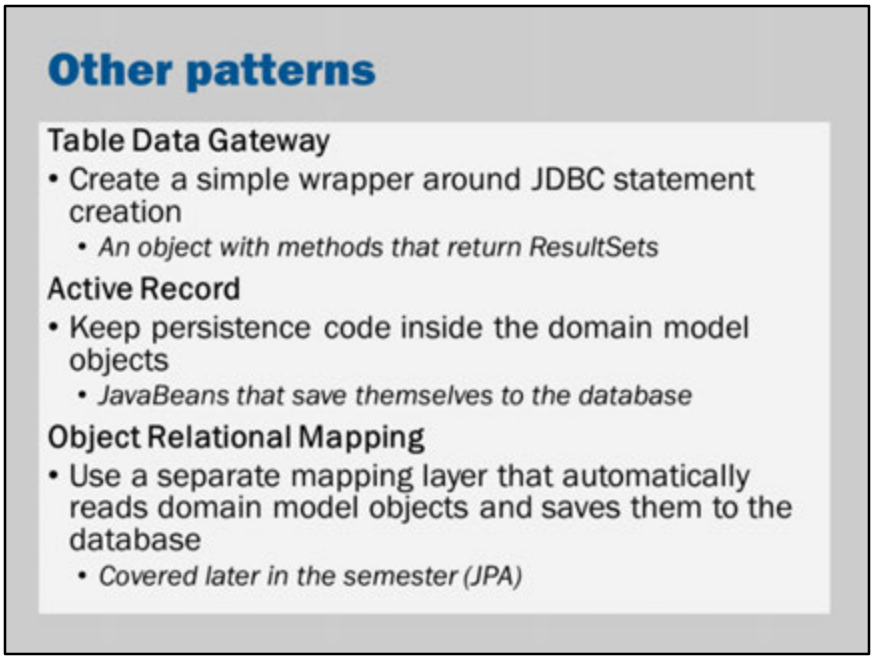
1. Client (Java code) gets an instance of the data access object
2. Client calls the read/find/get method on the DAO
3. DAO uses JDBC to locate record in database
4. DAO creates a DTO
5. DAO sets properties of DTO using data from database
6. DAO returns configured DTO to client



This is a UML sequence diagram that illustrates the flow of control (for a create/insert function on the DAO):

1. Client creates a DTO
2. Client sets properties of the DTO
3. Client passes DTO to DAO
4. DAO reads properties from DTO
5. DAO uses JDBC to store values into database

The DTO is just a simple container used to get data into (and out of) the DAO.



There are other techniques for separating persistence logic from domain logic.

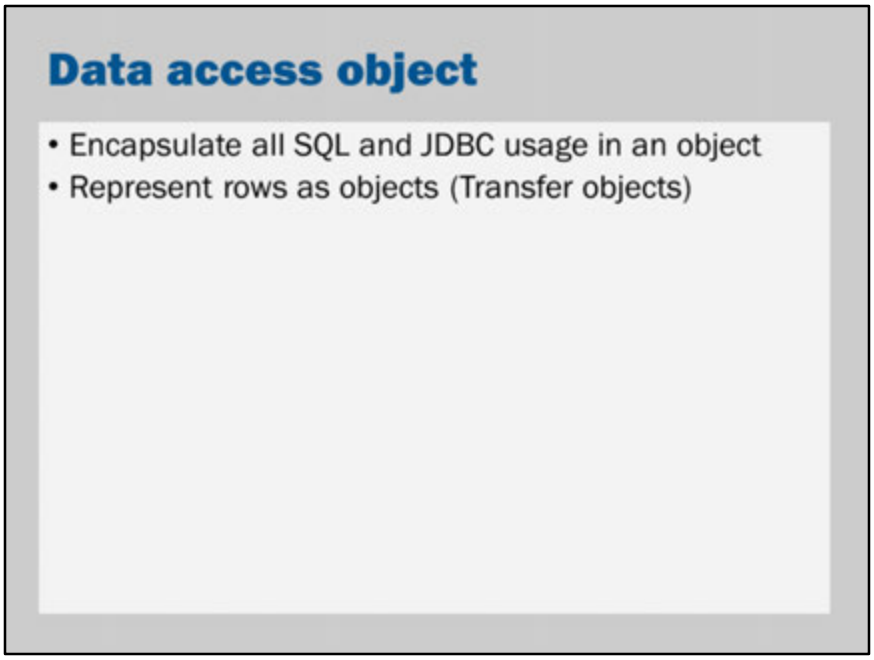
If you use the Table Data Gateway pattern, you would build a simple class (a gateway) that provides a wrapper around SQL.
The gateway would connect to the SQL database but return a ResultSet.
For example:

```
public class AccountTableGateway {
  public ResultSet find(int record) {
    Connection conn = …
    Statement stmt = …
    ResultSet rs = …
    return rs;
  }
  public void create(String username, String password, …) {
    Connection conn = …
    PreparedStatement ps = conn.prepareStatement("insert into …
    ps.setString(1, username);
    ps.setString(2, password);
    ps.execute();
```

```
    if (1 != ps.getUpdateCount()) {
      // handle error
    }
  }
}
```

An Active Record combines a DAO and DTO into a single file.
An Active Record is a DTO that has additional methods such as create and delete. When, for example, create is called, the Active Record will call the database itself and insert its own data into the database directly.

http://richard.jp.leguen.ca/tutoring/soen343-f2010/tutorials/implementing-table-data-gateway/
http://richard.jp.leguen.ca/tutoring/soen343-f2010/tutorials/implementing-active-record/



In fact, you have seen something like this before:
We had a "fake" database in the Week 4 tutorials that was essentially a DAO.
I say "essentially" because there is a slight difference.
The class in Week 4 had static methods.
A Data Access Object would be implemented using instance methods of a class.