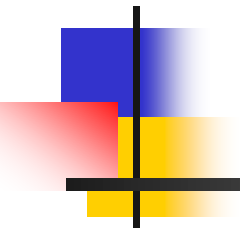
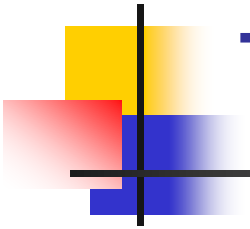


COMP1140: Database and Information Management

Lecture Notes – Week 12 (part B)

Dr. Suhuai Luo
School of EEC
University of Newcastle



This Lecture – part B

- You will learn:
  - Why we need to connect to databases from code
  - A re-fresh (or introduction) to Java Programming
  - What a Database Connector is
  - Security issues with database connectors
  - How to safely execute database queries from applications

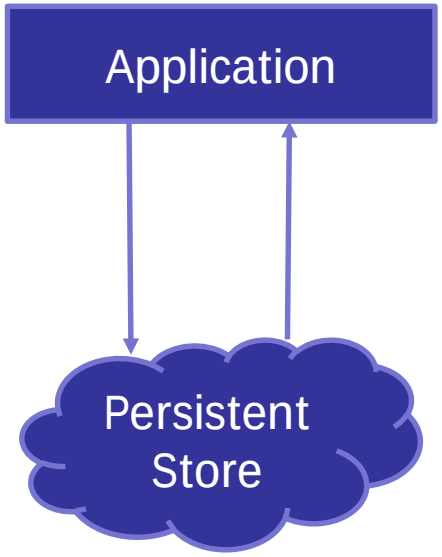
Databases in Applications

Applications store data in a Persistent Store

- Anything that persists data for long-term storage

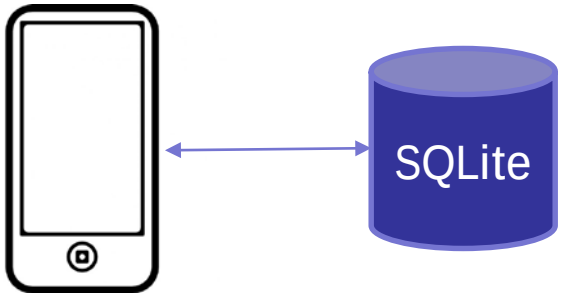
- Persistent stores have many forms
  - Can be a Relational Database (SQL)
  - Can be a Document Store (No-SQL)
  - Can be a Serialized file (e.g. XML, DOC, XLS, …)
  - Etc …

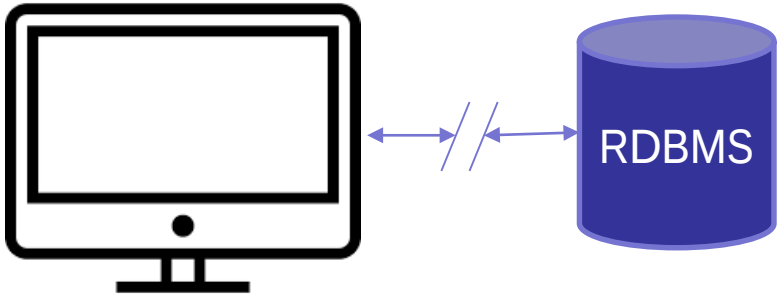
- The persistent store does not have to be local to the application
  - Can be managed by the application itself
  - Can be a DBMS on the local server
  - Can be on a remote server



Many Apps use Databases

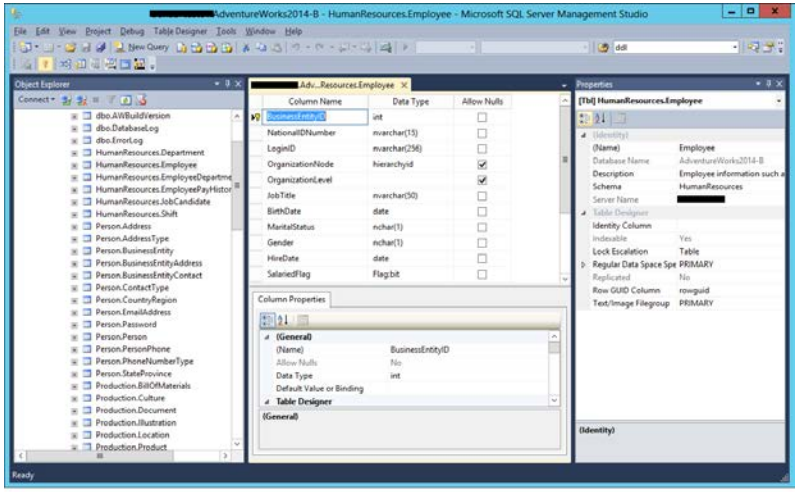
- All iOS, Android, Windows Phone apps have access to a database
  - Commonly SQLite (a file-based variant of SQL)
  - Store structured persistent application data
  - Common to see other persistence methods
- Most web applications use a RDBMS to store service data
  - Facebook, Google, LinkedIn, Twitter, …
  - Everything is backed by a RDBMS
- Desktop applications often use both of these approaches
  - Use a file to store local data
  - Store remote data in a RDBMS

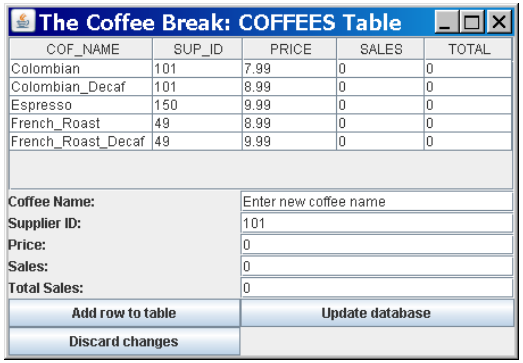


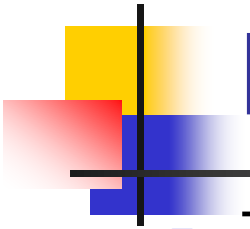


How to use a Database?

- MSSQL Management Studio can graphically develop a relational database schema
  - Create tables, relationships, constraints, etc …
  - Insert & edit data, etc …
- A database is useless without a program to use it.
- How do we utilize this from code?
- How do we execute queries in a program?

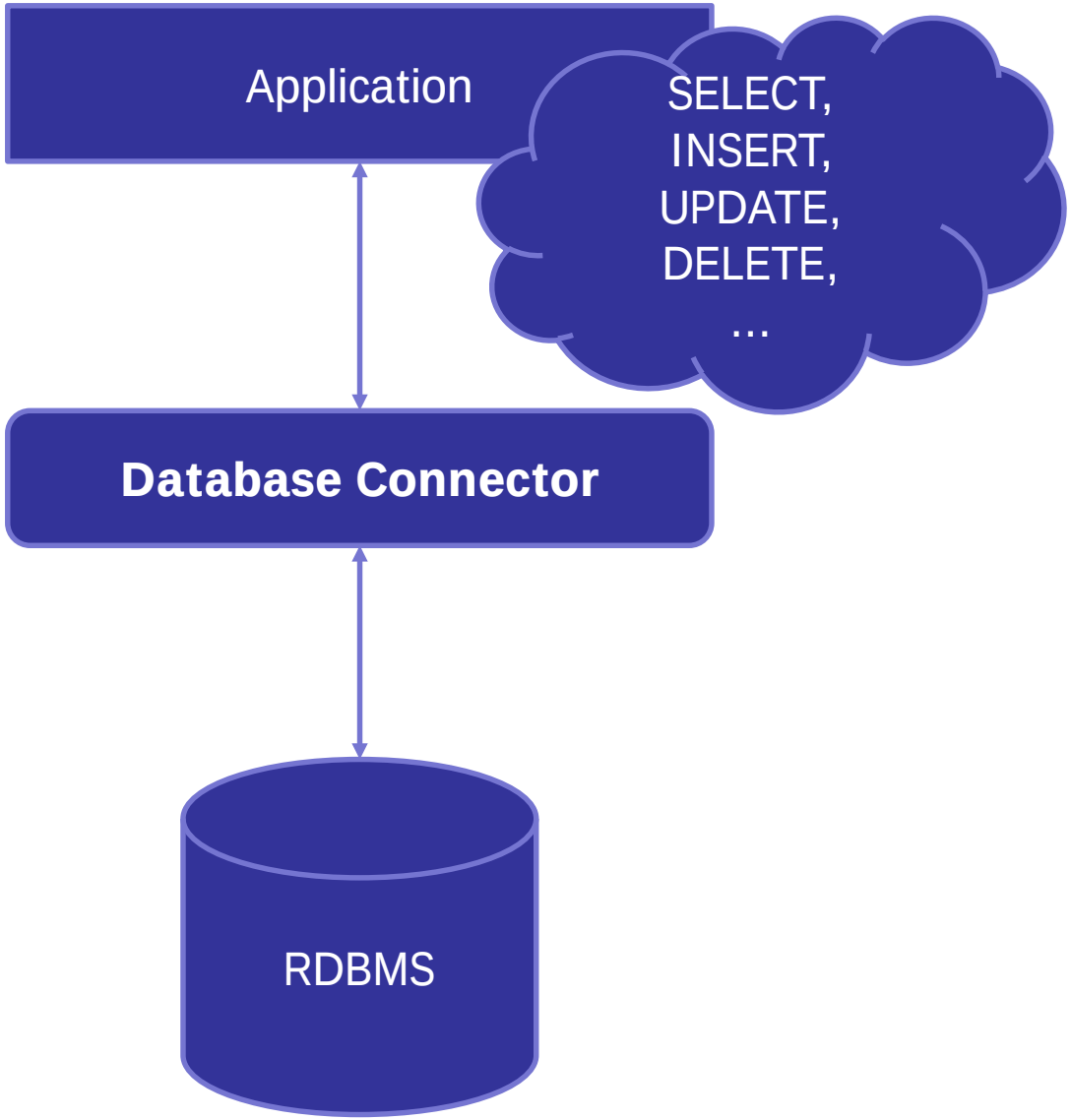






Database Connectors

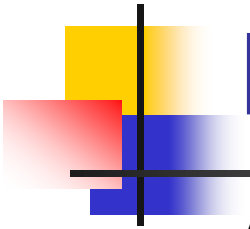
- They allow us to connect to a RDBMS from code and execute queries
  - SELECT queries to retrieve data
  - INSERT, UPDATE & DELETE to modify data
  - Call stored procedures
  - Even execute CREATE + MODIFY queries (unsafe)
- They are exposed as a set of Objects or Functions
  - Try to appear as native code
  - Handles errors as with native platform (e.g. Java exceptions)
- Bridges the native language types with SQL types
  - e.g. SQL varchar(255) -> Java String



An application which relies on a database and calls queries

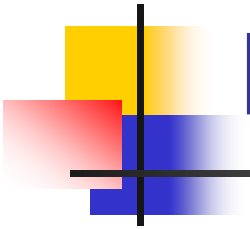
A Database Connector bridges the application and the RDBMS.
e.g. JDBC

A RDBMS storing persistent data for the Application



Database Connectors

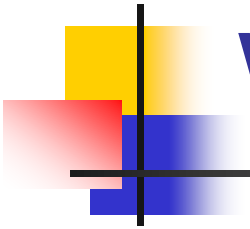
- All (good) programing languages will provide database connectivity with a framework
  - JDBC - > Java and JVM-based languages
  - ADO.NET -> C# and other .Net languages
  - ODBC -> C, C++, Java, Python, .Net (most languages)
- Most Database Vendors support using multiple connectors
- Many aim to be cross platform & database independent
  - On a mac I can use the same framework in the same manner as Linux, Unix, Windows ...
  - Same set of functions/methods used for any RDBMS
- In the Computer Labs we will be using **JDBC** to connect to a RDBMS
  - (Basic Java skills necessary)



Database Connector Process

- Generally, connecting is a three step process
- 1. Open a connection to the RDBMS
- 2. Execute a query
- 3. Handle result (or error)
- We will look further into this later with JDBC (Java)

Java Basics



What is Java?

- All students hopefully have done SENG1110 (Java) or INFT1004 (Python)
  - INFT1004 teaches Jython (Python on the Java Platform), close enough
- Java is a popular Object Oriented programming language
  - Cross platform, ‘Write Once Run Anywhere’
  - High level, easy to use, safe (too safe)
- Java code is strictly Object Oriented (with some exceptions)
  - Contains primitive types (int, double, char) -> Not Objects!
- Everything (without exception) is declared in a class
- Strongly typed -> References are declared to have a type which is strictly enforced

Classes

- Classes define objects, an object is an ‘instance’ of a class.
- A class will define the object’s ‘members’
  - Variables (data) and
  - Methods (functions/procedures)
  - Both have access modifiers (who can ’access’ a member)
- A class can also have its own fields and methods
  - Labelled as being ‘static’ in Java
  - Any other class or object can reference these without creating an instance
- Class ‘X’ is defined in file ‘X.java’ **-> Important!**

```
public class MyClass
{
  public static int classNum = 7;
  public static void classMethod()
  {
    // Do something important
  }

  public int instanceNum = 23;
  public void instanceMethod()
  {
    // Do something important
  }
}
```

Objects

- Objects are ‘instantiated instances’ of a class
- Created in Java with the ‘new’ operator
  - Calls a constructor
- Expose their fields and method for use by other objects/classes
- When defining a method, the current object is referred to be ‘this’ (only in non-static methods)

```
class MyClass {
  int instanceNum;

  public MyClass()
  {
    // Setup Class
    this.instanceNum = 3;
  }

  void instanceMethod()
  {
    // Do something important
  }
}

MyClass instance = new MyClass();
instance.instanceNum = 7;
instance.instanceMethod();
```

Access Protection

- Java promotes the use of Access Modifiers
- Restrict access to class + object members
  - Public – can be accessed anywhere
  - Private – can be access in the declaring class only
- By convention all variables should be private
  - Access through methods (getters and setters)

```
public class MyClass
{
   public static int classNum = 7;
   public static void classMethod()
   {
     // Do something important
   }

   private static int hiddenNum = 12;
   private static void hiddenMethod()
   {
     // Do something important
   }
```

Data Types

- Java offers many common data types
  - Integers – int, Integer
  - Floating point numbers – float, Float, double, Double
  - Booleans – boolean, Boolean
  - Characters – char, Character
  - Strings – String
- Java distinguishes between Object and Primitive types
  - Objects are instances of classes, contains members, can be null
  - Primitives are values, no class or members, cannot be null
  - Object types start with a capital, primitives lower case.
- Null is the absence of a value

```
int myInt = 7;

float myFloat = 1.140;
double myDouble = 1.1400;

boolean myBool = true;

char myChar = 'c';

String myString = "COMP1140";
Object anyObject = null;
```

Methods

- Methods perform an action and can return a value
- All functionality implemented in a method
- Built-in classes contain many methods
  - e.g. Append two strings, print to the console, read from the console.
- The entry point to a Java application is a static method named ‘main’

```
public class Main
{
  public static void main(String args[])
  {
    int num = 12;
    System.out.println(num);
  }
}
```

Basic IO

- Input:
  - Create a Scanner (reads from the console)
  - Get the next line (a String) or an int (or any primitive)
- Output:
  - System.out is an object that prints to the console
  - Two popular methods:
    - print and println

```
Scanner keyboard = new Scanner(System.in);
String line = keyboard.nextLine();
int num = keyboard.nextInt();


System.out.print("Print characters to the current line");
System.out.println("Print characters terminated by a newline");
```

Exceptions

- Not all methods can execute correctly all the time
- Java can declare methods to ‘thow’ exceptions
  - Methods can also throw undeclared RuntimeExceptions
- Exceptions represent errors which occur at runtime
  - Java will make sure you handle them out of safety
  - Code will sometimes not compile without error handling!
- E.g. try to divide by zero
  - a DivideByZero exception will be thrown

```
public void myUnsafeMethod() throws Exception
{
  // Do some stuff

  if (error occured)
  {
    throw new Exception("An error occured");
  }
}
```

```
public void myBadMathsMethod()
{
  try {
    int x = 4 / 0.0;
  }
  catch (DivideByZeroException e) {
    System.out.println("Cannot divide by zero!");
  }
}
```

Compiling + Running

- Java is a compiled language, but runs on a virtual machine (for cross-platform)

- 1. Compile X.java
- “javac X.java” (or “javac *.java” for multiple files)

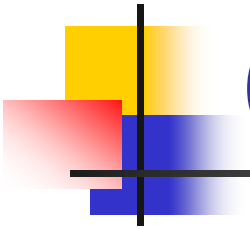
- 2. Run class X
- “java X”
- Where X contains a static method ‘main’ and is a Java class

```
class MyClass {
  public static void main(String args[])
  {
    System.out.println("Hello, World");
  }
}
```

```
[Kronos:~ haydencheers$ javac MyClass.java
[Kronos:~ haydencheers$ java MyClass
 Hello, World
```

JDBC

Java Database Connectivity (JDBC)



- Provides mechanisms to:
  - Configure a database
  - Connect to a database
  - Construct and execute queries
  - Process results
  - Disconnect from the database
- Can interact with any SQL database
  - As long as the database vendor supports it

JDBC

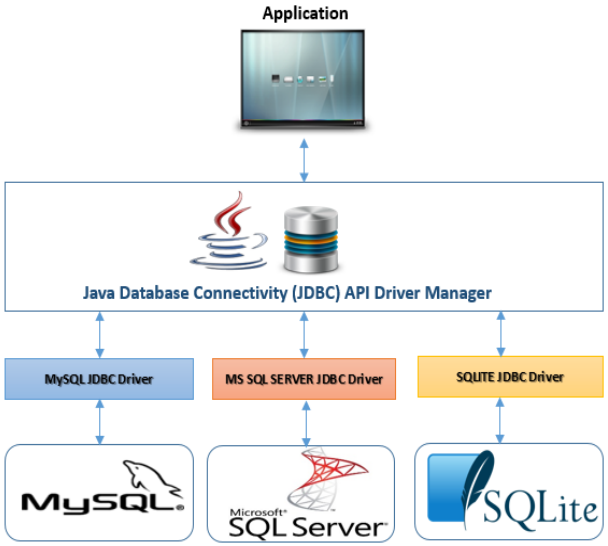
- Provides a set of classes to allow any Java application to connect to a database
  - Provided with a standard Java Runtime Environment (JRE) installation
  - On every supported platform (i.e. cross platform)

- Helps make it possible to write a single java application that can access a range of DBMS

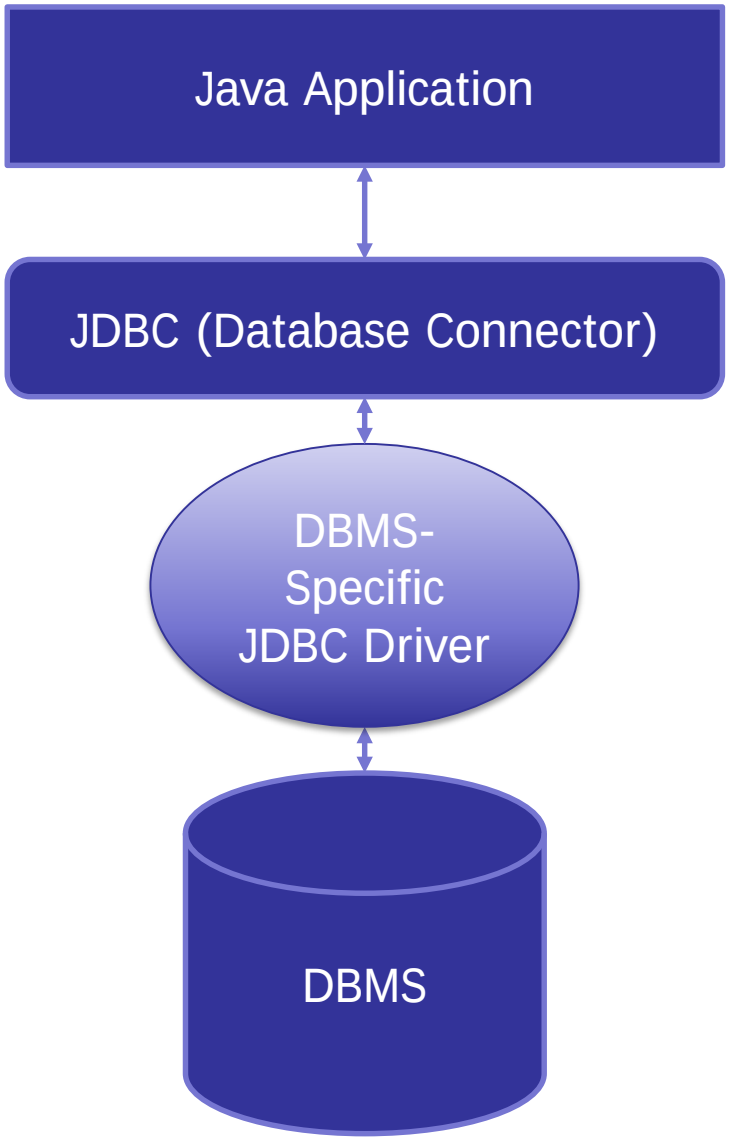
JDBC

- Can be used in any type of Java application
  - Command line
  - Graphical
  - Web
  - Even Android! (NOT RECCOMENDED, better ways of doing this)

- Requires a vendor-specific driver
  - As we use MSSQL, we need the MSSQL JDBC Driver
  - Bridges the gap between Java and the RDBS
  - JDBC abstracts the actual connection to a DB, relying on the Driver
  - Allows JDBC code to be reusable regardless of the database server used
  - This works in theory, but SQL servers use different SQL variants!



Similar architecture to a generic database connector



Any Java application

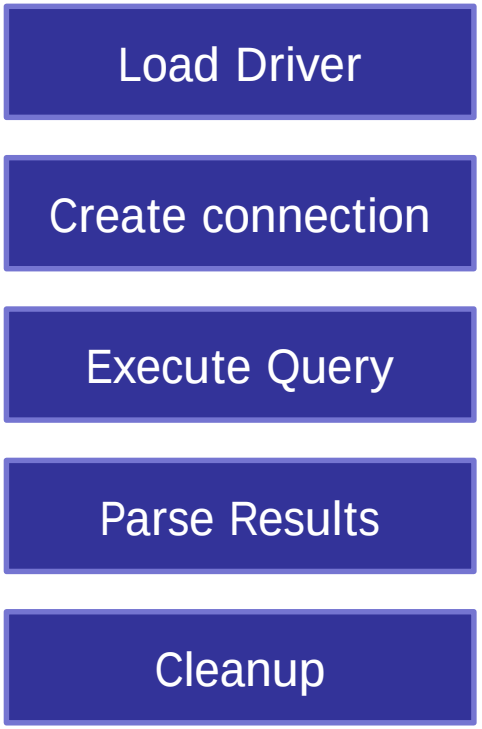
The JDBC Framework

A JDBC has a vendor-specific driver which knows how to communicate with the DBMS (eg MSSQL Driver)

A DBMS storing persistent data for the Application (eg MSSQL)

JDBC

- In a standard Java application we need to manage every aspect of the database communication
  - Load the driver at runtime
  - Create a connection + authenticate
  - Create + execute queries
  - Parse results
  - Close the connection
- All JDBC related code needs to be wrapped in try/catch
  - Exceptions can occur at any step in this process!
- Java web applications automate this process for efficiency
  - We learn about this in SENG2050



Using JDBC – Step 0 (Setup)

- Create a Java class

```
import java.sql.*;

public class Main
{

}
```

- Import java.sql.*;
  - This exposes the JDBC classes

- Find a copy of the JDBC Driver!
  - Needs to be in the classpath (we do this in the lab)
  - Download from: https://docs.microsoft.com/en-us/sql/connect/jdbc/microsoft-jdbc-driver-for-sql-server

Using JDBC – Step 1

- Load the vendor-specific JDBC Driver at runtime
  - This can be done anywhere, as long as it occurs before we create the first connection
  - We need to know the fully-qualified name of the driver (i.e. the name & package of the class)
  - We load Java classes at runtime with the Class.forName() method, which will try to find the class
  - This can throw an exception (we need to wrap it in try/catch in practice)

```
public class Main
{
  public static void main(String args[])
  {
    // Load the driver
    Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver").newInstance();
  }
}
```

Using JDBC – Step 2

- Create a connection to the DBMS
  - Need to know the DBMS connection string
    - Generally need to know the server URL, database name, a username and password
    - Format is vendor specific!
  - If we change the database vendor, this is the only step that needs to change (in theory)
- Connection is made by the DriverManager class
  - Pass it the connection string (and optionally a username + password)
  - If we can't connect to the server, an exception is thrown

```
// For MSSQL
String mssqlURL = "jdbc:sqlserver://<host>;databaseName=<database>;
                                                    user=<username>;
                                                    password=<password>";
// For MySQL
String mysqlURL = "jdbc:mysql://<host>/<database>";
// Username and password are passed seperately
```

```
String url = "<connection string>";

Connection connection = DriverManager.getConnection(url);
// OR
DriverManager.getConnection(url, "username", "password");
```

Using JDBC – Step 3

- Create a JDBC Statement

- A statement is an object that executes a query
  - We define a query and the object will execute it
  - The Statement will return a result based on the type of query
  - Creating a statement can throw an exception, needs to be in try/catch!

```
Statement statement = connection.createStatement();
```

Using JDBC – Step 4a

- Execute the query
  - Depending on the type of query, we decide how we execute the statement
    - For SELECT queries:

```
String query = "SELECT * FROM MyTable;";
ResultSet results = statement.executeQuery(query);
```

    - For all others:

```
String query = "INSERT INTO MyTable (...) VALUES (...)";
int result = statement.executeUpdate(query);
```

Using JDBC – Step 4b

- Parse the results

- Results are returned as a ‘table’
  - Rows are records, columns are attributes
  - We iterate through the rows and retrieve attributes by column name
  - Have to specify the expected type of the data
  - Can throw exceptions if invalid column name or incorrect type

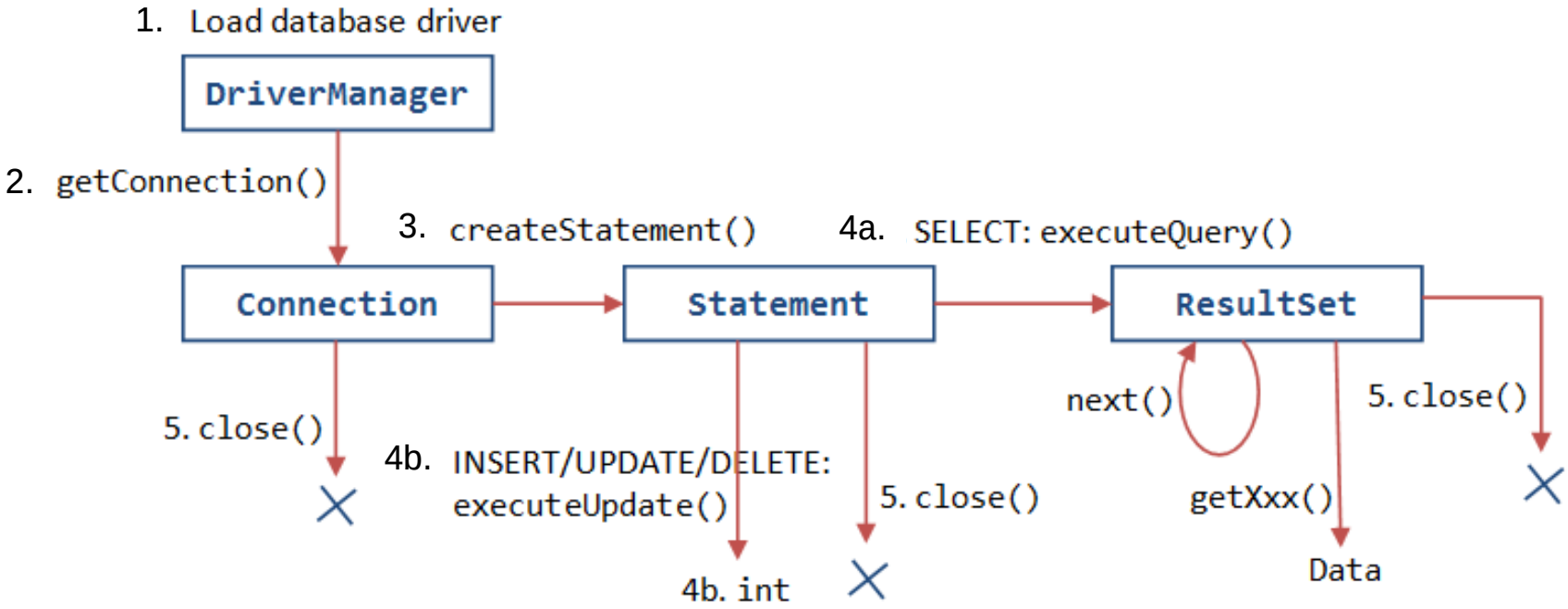
id	amount	name	flag	date
1	$100.00	Item 1	TRUE	16/7/17
2	$150.00	Item 2	FALSE	17/7/17
3	$75.00	Item 3	FALSE	19/7/18

```
ResultSet results = statement.executeQuery(query);
while (results.next())
{
  int id = results.getInt("id");
  float amount = results.getFloat("amount");
  String name = results.getString("name");
  boolean bool = results.getBoolean("flag");
  DateTime datetime = results.getDateTime("date");
  ...
}
```

Using JDBC – Step 5 (Cleanup)

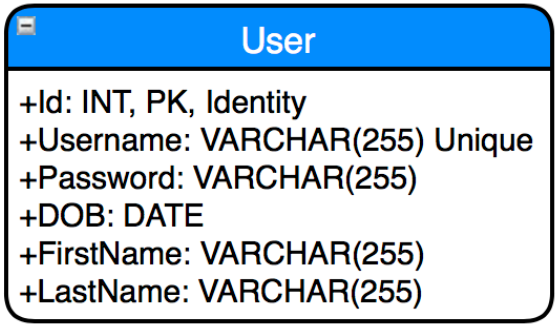
- Finally, we need to close the connection, statement and results
  - Results must be closed before we execute another query
  - Statements must be closed before we create another query
  - You can open as many connections as required however
  - But this is expensive!

```
results.close();
statement.close();
connection.close();
```

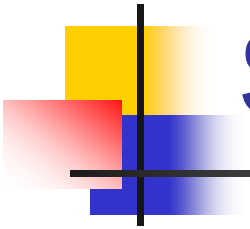

1. Load database driver
DriverManager
2. getConnection()
3. createStatement()
4a. SELECT: executeQuery()
Connection
Statement
ResultSet
5. close()
4b. INSERT/UPDATE/DELETE:
executeUpdate()
5. close()
next()
5. close()
getXxx()
4b. int
Data

JDBC Code Sample

Selecting all birthdays from a ‘Users’ table

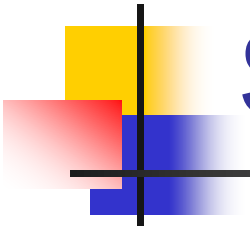



```java
import java.sql.*;  // JDBC classes
import java.util.*; // List + LinkedList

public class MyQuery {
  public static void main(String[] args) {
    try {
      // Step 1 - Load Driver
      Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver")
           .newInstance();

      // Step 2 - Create Connection
      String url = "jdbc:sqlserver://<host>;databaseName=<database>;"
                 + "user=<user>;password=<password>";

      Connection connection = DriverManager.getConnection(url);

      // Step 3 - Create Statement
      String query = "SELECT DISTINCT [DOB] FROM [User]";
      Statement statement = connection.createStatement();

      // Step 4 - Execute query + parse results
      List<Date> dates = new LinkedList<>();
      ResuleSet results = statement.executeQuery(query);
      while (results.next()) {
        Date date = results.getDate("DOB");
        dates.add(date);
      }

      // Step 5 - Cleanup
      results.close();
      statement.close();
      connection.close();

      // Print Dates
      for (Date data : dates) {
        System.out.println(date);
      }
    } catch (Exception e) {}
  }
}
```



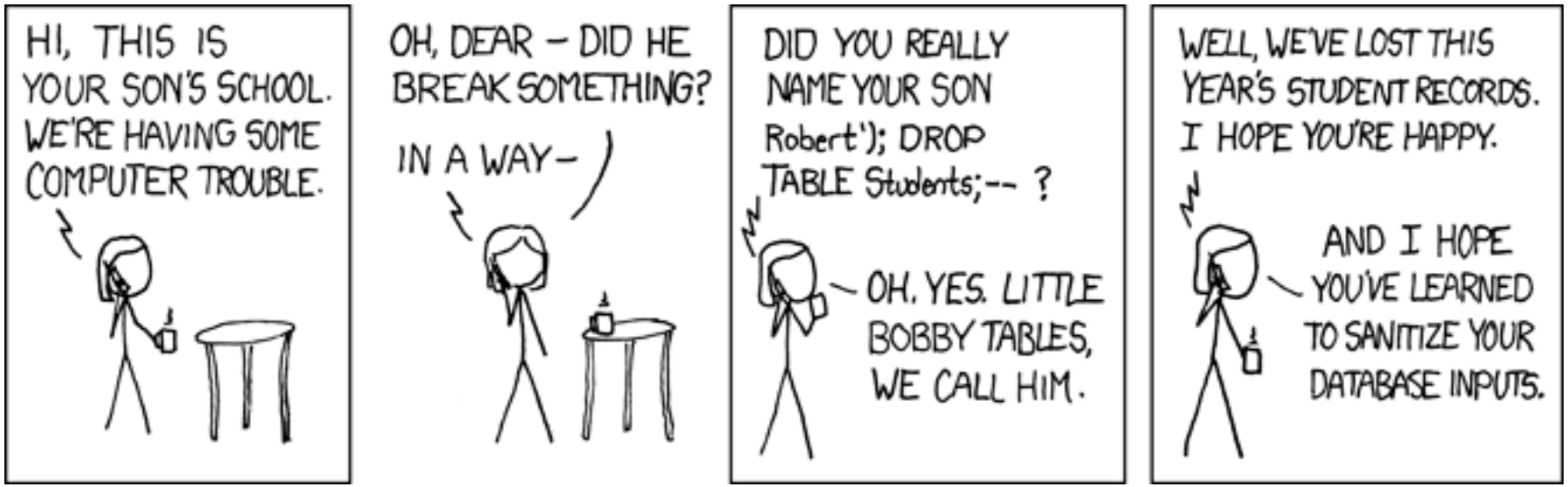
Safe Queries

- In any application user-input is potentially malicious
  - ‘Hacking’ attempts
- Need to know what to expect
  - If we expect an int, we need to make sure its an int
  - Range checking
  - Length checking
  - Sanitize input
  - ‘Escape’ unsafe input
- What if we construct a SQL query with user parameters?
  - What could happen if we don’t check user input?



SQL Injection

- Poorly constructed database queries are susceptible to SQL Injection
- A malicious user can inject malicious SQL into a un-sanitized query
- Common attack approach to web applications
  - Potential to occur wherever there is a database interaction with user input
- Can be used to
  - **Access application without privilege**
  - Insert malicious data
  - **Drop tables**
  - Steal data
  - Numerous …


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

SQL Injection Example

- Let's have a table named 'User' with columns
  - Username
  - Password
- An application requests two user-supplied fields from the console
  - Username
  - Password

```
Scanner keyboard = new Scanner(System.in);

String username = keyboard.nextLine();
String password = keyboard.nextLine();
```

- If we don't check this input, it could be anything! (EVEN SQL)

SQL Injection Example

- To validate a username and password, lets use a query

```
String query = "SELECT TOP 1 FROM [User] WHERE [Username] = '"
             + username + "' AND [Password] = '" + password + "'";
```

- If we supply a valid username + password, we retrieve a single record
  - Otherwise nothing is returned

```
"SELECT TOP 1 FROM [User] WHERE [Username] = 'suhuai'"
+ "AND [Password] = '1234'";
```

- **What if the username and password are not plain text?**

SQL Injection Example 1

- Malicious user supplies bad input
  - Username = “hackerman”
  - Password = “1234’ OR 1=1 --”

- SQL Query will be constructed as

```
"SELECT TOP 1 FROM [User] WHERE [Username] = 'hackerman'"
  + "AND [Password] = '1234' OR 1=1 --'";
```

- What have we done?
  - **Query will return a record for a random user -> query finds a ‘matching’ user**
  - This query will always find a match (if there are users in the database)

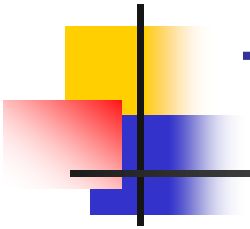
SQL Injection Example 2

- Malicious user supplies **REALLY** bad input
  - Username = “hackerman”
  - Password = “1234’; DROP TABLE [User]; --”
- SQL Query be constructed as

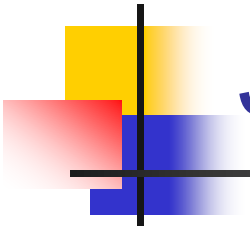
```
"SELECT TOP 1 FROM [User] WHERE [Username] = 'hackerman'"
  + "AND [Password] = '1234'; DROP TABLE [User] --'";
```

- Two Queries!
  - First will try to select the user
  - **Second will DROP THE User Table!**

How can we protect against this?



- Escaping or sanitizing user input
  - i.e. removing or escaping any SQL grammar from the input
- Pattern Checking
  - Using regular expressions to ensure input doesn't match SQL
- Database Permissions
  - Ensure the user connecting to the DBMS cannot perform malicious operations
    - i.e. Can't drop tables
- **Prepared/Parameterized Queries**
  - Define a 'skeleton' of the SQL query with parameter placeholders



JDBC Prepared Statements

- JDBC offers PreparedStatements (as opposed to a Statement)
  - Define SQL query ‘skeleton’ with parameter placeholders
  - Prepares Statement (Validates query on server, server knows what to expect)
  - Bind statement parameters
  - Execute query

- Sanitizes statement parameters
  - Removes any SQL
  - Stops SQL Injection

Using Prepared Statements

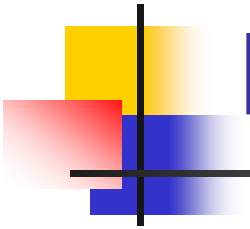
JDBC Statement

```
String query = "SELECT FROM ... WHERE ... = " + param;
Statement stmt = connection.createStatement();
stmt.executeQuery(query);
```

JDBC PreparedStatement

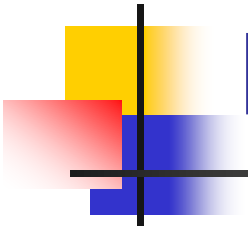
```
String query = "SELECT FROM ... WHERE ... = ?";
PreparedStatement stmt = connection.prepareStatement(query);
stmt.setInt(1, param);  // Method name is type based
stmt.executeQuery();
```

- **Safer**
- Small change in usage
- Query is defined, prepared and parameters are bound



Benefits

- No chance of SQL Injection
  - When used correctly
  - DBMS (usually) parses the query to ‘know’ how it should be executed
  - i.e. query table x on columns y and z -> No other queries or comparisons should be present
  - Parameter values are sanitized
- Query can be optimized and cached by the server (increased performance, if supported)
- **ALWAYS USE PREPARED STATEMENTS WHEN PARAMETERS ARE PRESENT!**



Review

- Applications use databases for persistence
- Use Database Connector to communicate with database
- DB Connectors bridge the SQL with a Programming Language
- ~3 Step process to use
- Susceptible to SQL injection attacks
- Use ‘Parameterized’ or ‘Prepared’ queries to protect against this
- **ALWAYS USE PREPARED QUERIES!**