Week 2 Review

Data Definition Language (DDL)

- **CREATE/DROP DATABASE**
- **CREATE/DROP TABLE/DROP TABLE IF EXISTS**
- **ALTER TABLE**
  - **ADD/DROP CONSTRAINT PRIMARY KEY**
  - **ADD/DROP CONSTRAINT FOREIGN KEY**
  - **ADD/DROP CONSTRAINT CHECK**
- **CREATE/DROP SEQUENCE**
  - **SERIAL DATA TYPE IN POSTGRES**
  - **NEXTVAL**

Data Control Language (DCL)

- **GRANT**
- **REVOKE**
- **Separate permissions for**
  - **SELECT**
  - **INSERT**
  - **UPDATE**
  - **DELETE**

First Normal Form (1NF)

- No duplicative columns
- Every table organized in rows with primary keys that uniquely identify it

Customer	Items
Dave Buster	Scissors, Tape, Glue
Molly McButter	8 x 10 Frame, Framing Mat
Dave Buster	Large Box

Customer Item ID	Customer	Item
1	Dave Buster	Scissors
2	Dave Buster	Tape
3	Dave Buster	Glue
4	Molly McButter	8 x 10 Frame
5	Molly McButter	Framing Mat
6	Dave Buster	Large Box

Second Normal Form (2NF)

- 1NF
- Non-key attribute must be dependent on the primary key.

Course ID	Instructor ID	Course Name
1	1	Java
2	3	.NET
1	2	Java

➡️

Course ID	Course Name
1	Java
2	.NET

Course ID	Instructor ID
1	1
2	3
1	2

Third Normal Form (3NF)

- 2NF
- No transitive functional dependency.

Category	Year	Winner	Winner DOB
Best Solo Performance	2020	Lizzo	4/27/1988
Record Of The Year	2020	Childish Gambino	9/25/1983
Best Solo Performance	2019	Lady Gaga	3/28/1986
Record Of The Year	2019	Childish Gambino	9/25/1983

Category	Year	Winner
Best Solo Performance	2020	Lizzo
Record Of The Year	2020	Childish Gambino
Best Solo Performance	2019	Lady Gaga
Record Of The Year	2019	Childish Gambino

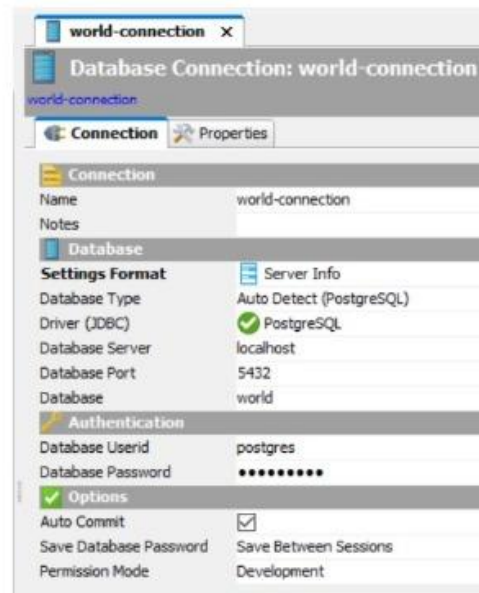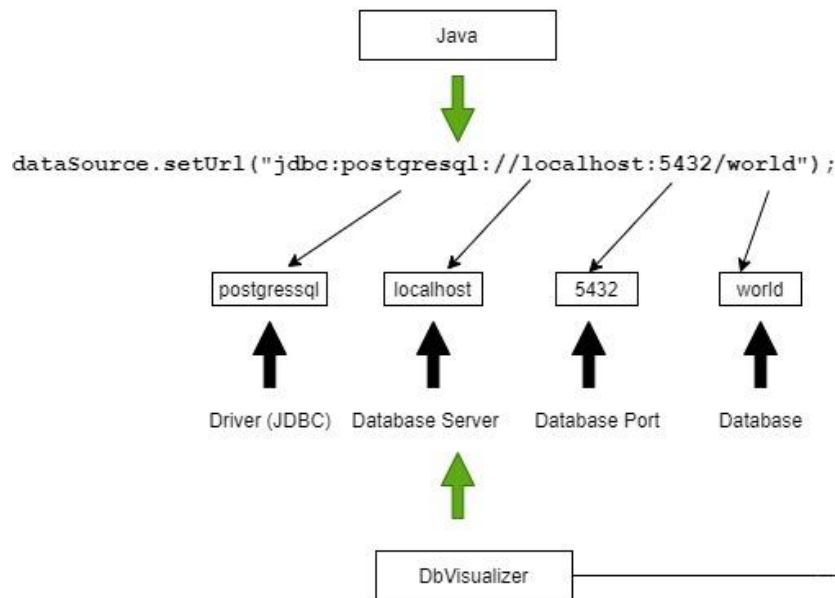| Winner	Winner DOB
Lizzo	4/27/1988
Lady Gaga	3/28/1986
Childish Gambino	9/25/1983

Managing Connections

When we interact with a database, we need to **create a connection**.

- Connections **remain open** until they are closed or time out.

- Connections have overhead when created and opened, thus there is often a **finite number of connections**.

- A **connection pool** can be used to **reuse a few connections** to conserve resources within an application by allowing the application to **acquire** a connection and **release** it when it is no longer needed so it can be reused.
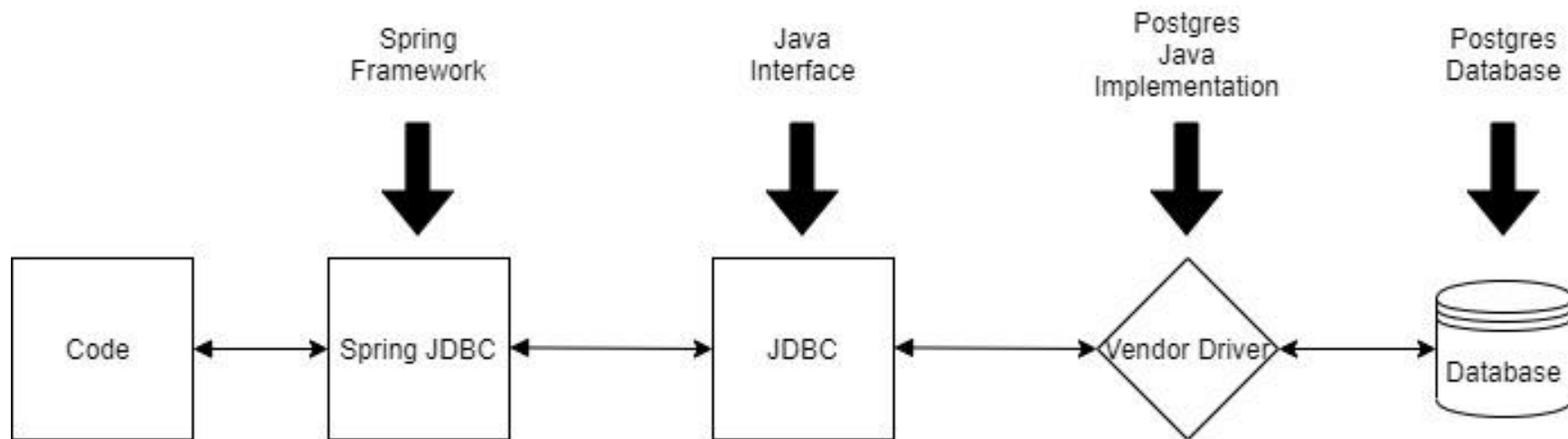
Managing Connections

Java Example:

```java
BasicDataSource dataSource = new BasicDataSource();
dataSource.setUrl("jdbc:postgresql://localhost:5432/world");
dataSource.setUsername("postgres");
dataSource.setPassword("postgres1");
```

Java

```java
dataSource.setUrl("jdbc:postgresql://localhost:5432/world");
```

| postgressql | localhost | 5432 | world |

Driver (JDBC)   Database Server   Database Port   Database

DbVisualizer

---

world-connection ✕

Database Connection: world-connection

world-connection

🔵 **Connection**   🔧 Properties

Connection
| Name | world-connection |
| Notes | |

Database
Settings Format	Server Info
Database Type	Auto Detect (PostgreSQL)
Driver (JDBC)	✅ PostgreSQL
Database Server	localhost
Database Port	5432
Database	world

Authentication
| Database Userid | postgres |
| Database Password | •••••••• |

✅ **Options**
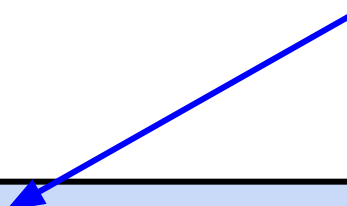| Auto Commit | ☑ |
| Save Database Password | Save Between Sessions |
| Permission Mode | Development |

Spring JDBC

JdbcTemplate

- **queryForRowSet** is used to SELECT data sets from the database

- **queryForObject** is used to SELECT a single value from the database

- **update** is used for modifying data in the database
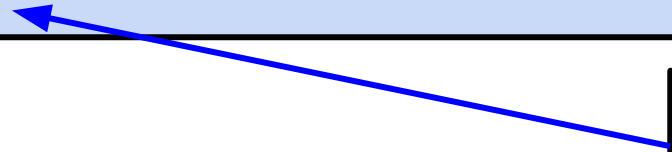   - INSERT
    - UPDATE
    - DELETE

Creating a JdbcTemplate

```
// Create BasicDataSource (remember that diagram?)
BasicDataSource dataSource = new BasicDataSource();
dataSource.setUrl("jdbc:postgresql://localhost:5432/world");
dataSource.setUsername("postgres");
dataSource.setPassword("postgres1");

// Create a JdbcTemplate using the Datasource
private JdbcTemplate jdbcTemplate =
    new JdbcTemplate(dataSource);
```

Create `JdbcTemplate` with the `datasource` as the parameter

Selecting Data with JdbcTemplate

```
SqlRowSet results = jdbcTemplate.queryForRowSet("SELECT
name, countrycode FROM city");
```

```
// with query params
SqlRowSet results =
    jdbcTemplate.queryForRowSet("SELECT id, name, countrycode
    FROM city WHERE id = ?", id);
```

Selecting Data with JdbcTemplate

```
Integer result = jdbcTemplate.queryForObject("SELECT
    COUNT(*) FROM city", Integer.class);
```

```
// with query params
Integer result = jdbcTemplate.queryForObject("SELECT
    COUNT(*) FROM city WHERE id = ?", Integer.class, id);
```

Getting Data From SqlRowSet

```
while(results.next()) {
    String name = results.getString("name");
    Long id = results.getLong("id");
    System.out.println(id + " " + name + ")");
}
```

Updating Data with JdbcTemplate

```
jdbcTemplate.update("UPDATE city SET population=600000
WHERE id=3825");
```

```
// with query params
jdbcTemplate.update("UPDATE city SET population=600000
    WHERE id= ? ", id);
```

Following the DAO Pattern

The **DAO pattern** uses a **data access interface** to add an abstraction layer to data objects. The pattern consists of:

- Data Access Object Interface

- Data Access Implementation Class

- Model (or Value) Objects - the DTOs or POJOs we mentioned previously.

Integration Testing

Integration Testing is a broad category of tests that validate the integration between units of code, or between code and outside dependencies such as databases or network resources.

Integration tests:
- Use the same tools as unit tests (i.e. JUnit)
- Usually slower than unit tests (but often still measured in ms)
- More complex to write and debug
- Can have dependencies on outside resources like files or a database.

DAO Integration Testing

Since DAOs exist solely for the purpose of interacting with a database, they are often best tested using an integration test.

Integration tests with a database ensure DAO code functions correctly:

- **SELECT** statements are **tested by inserting dummy data** before the test
- **INSERT** statements are **tested by searching for the data**
- **UPDATE** statements are **tested by verifying dummy data changed** or that rowsAffected == expectedResult
- **DELETE** statements are tested by seeing if **dummy data is missing** or that rowsAffected == expectedResult

DAO Integration Testing

Tests (including integration tests) should be:

- **Repeatable:**
  - If the test passes/fails on first execution, it should pass/fail on second execution if no code has changed.
- **Independent:**
  - A test should be able to be run on it's own, independently of other tests, OR together with other tests and have the same result either way.
- **Obvious:**
  - When a test fails, it should be as obvious as possible why it failed.

Approaches for Managing Test Data

- Remotely Hosted Shared Test Database

- Locally Hosted Test Database

- Embedded, In-memory Database

Using Transactions For DAO Integration Testing

In order to allow us to rollback transaction, we `setAutoCommit(false)`. This prevents the database from automatically committing after each SQL statement executes. Since queries are not being auto-committed, we can roll them back when our tests complete to avoid permanently affecting data in the database.

The **@BeforeClass** annotation denotes code to be called before the test class is created.
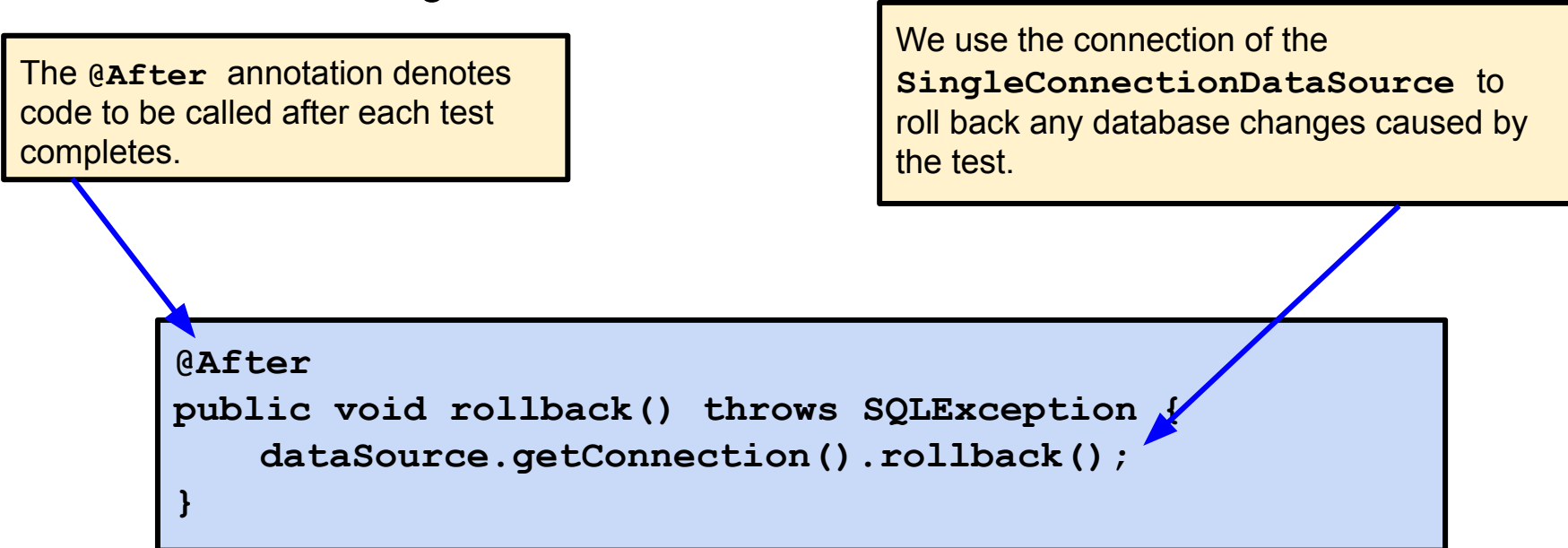
We call setAutoCommit(false) here.

```
@BeforeClass
public static void setupDataSource() {
    dataSource = new SingleConnectionDataSource();
    dataSource.setUrl("jdbc:postgresql://localhost:5432/world");
    dataSource.setUsername("postgres");
    dataSource.setPassword("postgres1");
    dataSource.setAutoCommit(false);
}
```

Using Transactions For DAO Integration Testing

After each test we use the **SingleConnectionDataSource** connection to roll back all changes.

The **@After** annotation denotes code to be called after each test completes.

We use the connection of the **SingleConnectionDataSource** to roll back any database changes caused by the test.

```
@After
public void rollback() throws SQLException {
    dataSource.getConnection().rollback();
}
```

Using Transactions For DAO Integration Testing

SingleConnectionDataSource is useful because it does not close connections automatically after it used. This allows us to roll back changes after each test but it also requires us to destroy the connection manually when we are done using the test class.

The **@AfterClass** annotation denotes code to be called before the test class is destroyed

In the **@AfterClass** code we destroy the **SingleConnectionDataSource**

```
@AfterClass
public static void closeDataSource() throws
SQLException {
    dataSource.destroy();
}
```

How Does SQL Injection Work?

```
String query = "SELECT population from CITY WHERE city_id = "
 + cityId;
```

Valid data:

```
SELECT population from CITY WHERE city_id = 237;
```

Dangerous data:

```
SELECT population from CITY WHERE city_id = 237 OR 1 = 1;
```

Preventing SQL Injection

- **Parameterized Queries**
  - The single most effective thing you can do to prevent SQL injection is to use parameterized queries. If this is done consistently, SQL injection will not be possible.
- **Input Validation**
  - Limiting the data that can be input by a user can certainly be helpful in preventing SQL Injection, but is by no means an effective prevention by itself.
- **Limit Database User Privileges**
  - A web application should always use a database user to connect to the database that has as few permissions as necessary. For example, if your application never deletes data from a particular table, then the database user should not have permission to delete from that table. This will help to limit the damage in the case that there is a SQL Injection vulnerability.

Dealing With Passwords

- We need to be able to **verify** a password but not **recover** it.

- A system administrator with access to credential data should not be able to determine a password.

- Any hacker that steals a database or set of credentials should not be able to read the passwords.

- Even with super-computing capabilities, no one should be able to access the data within any reasonable amount of time.
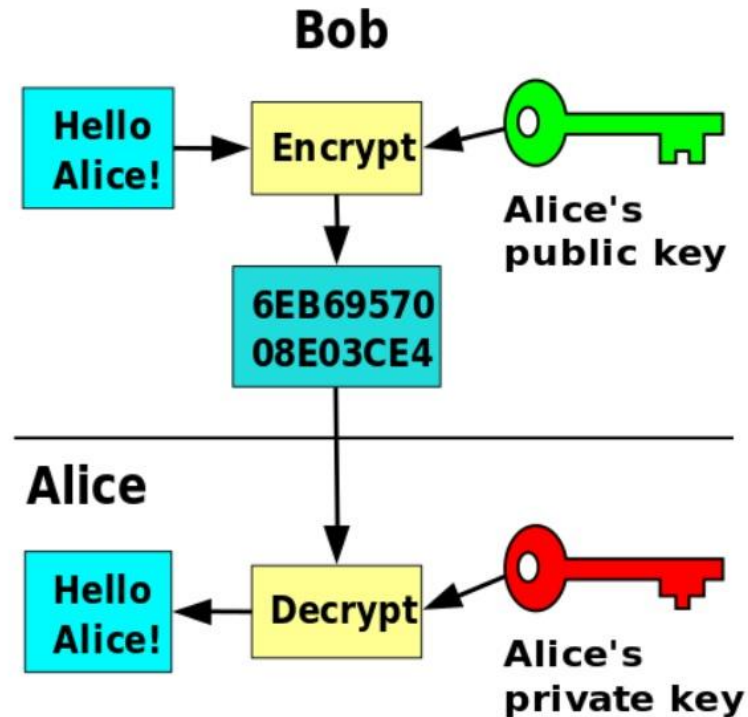
Hashing

Hashing process:

- Use a **one-way function to obfuscate** the plain-text password prior to storage.
- Use the password supplied by the user, **re-hash** it, and **/**it to the stored password hash value.
- **Salt the passwords** in order to make it take longer to calculate all possibilities.

Symmetric Encryption
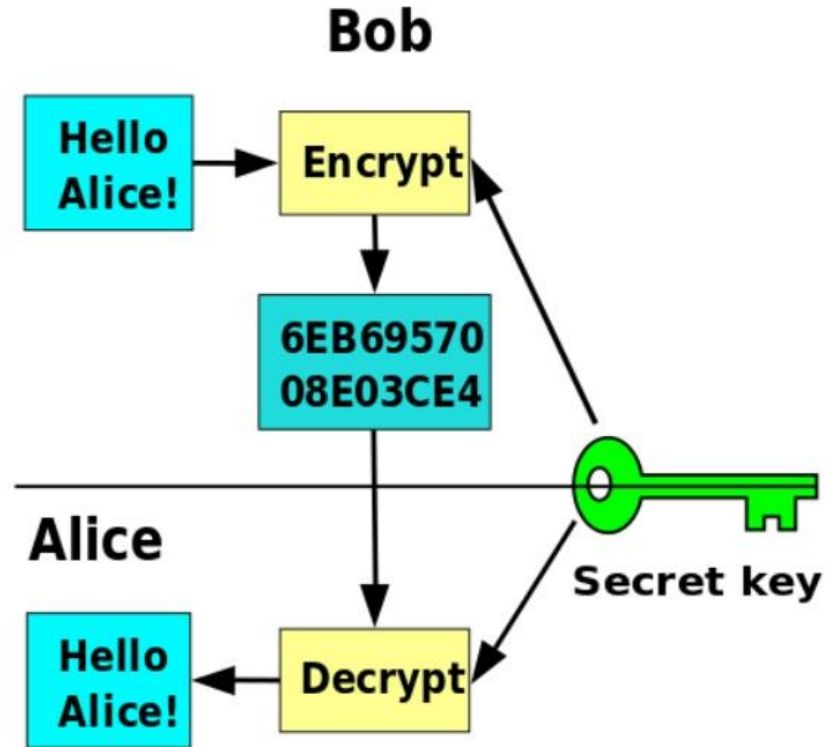
Asymmetric Encryption

Encryption on the Web

Web encryption

- Secure Socket Layer (**SSL**) and Transport Layer Security (**TLS**) are examples of **asymmetric key encryption**.

- Digital Certificates and Certificate Authorities (**CA**)

- Man-in-the-Middle Attack

.