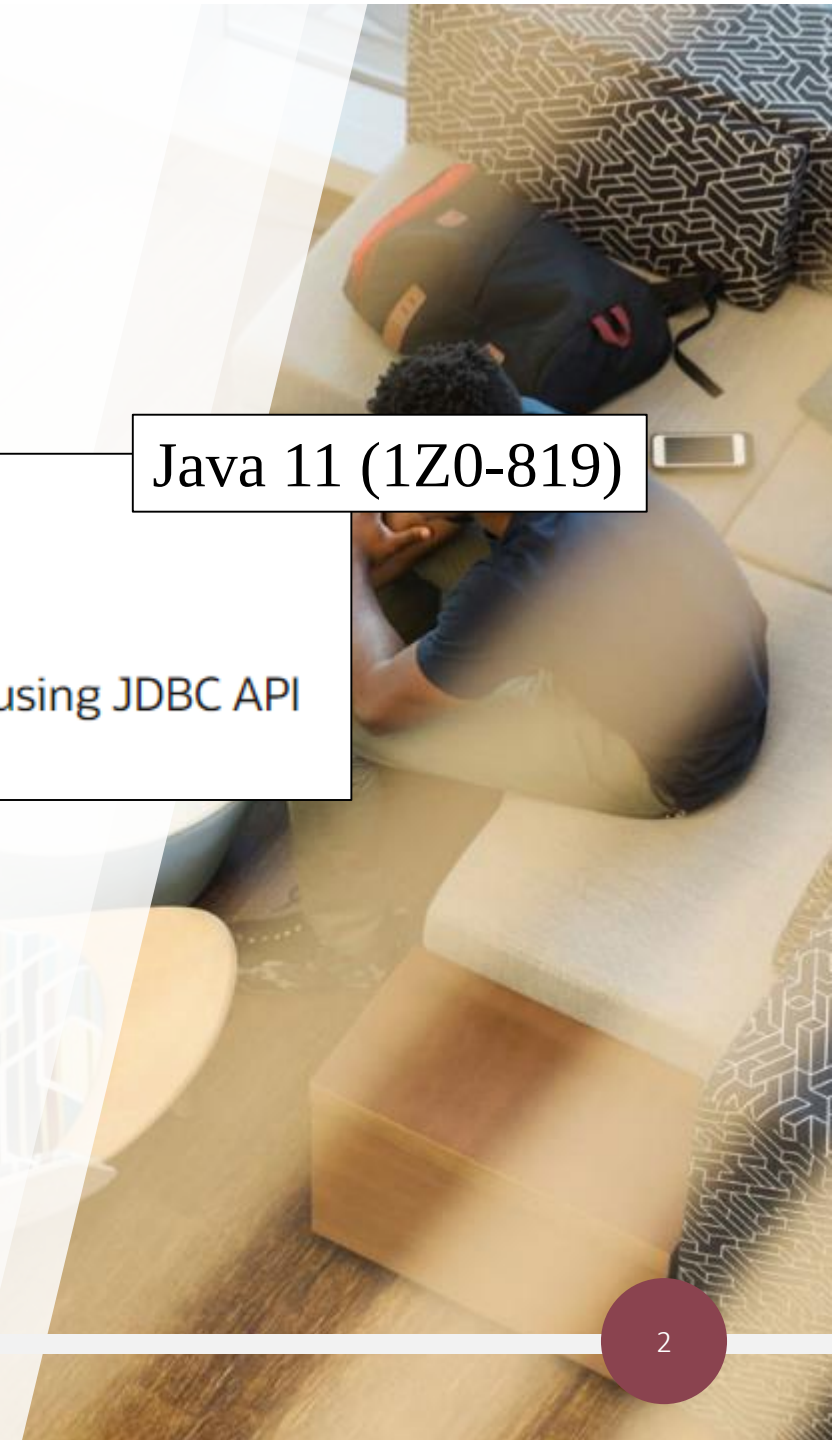
JDBC

JDBC

Java 11 (1Z0-819)

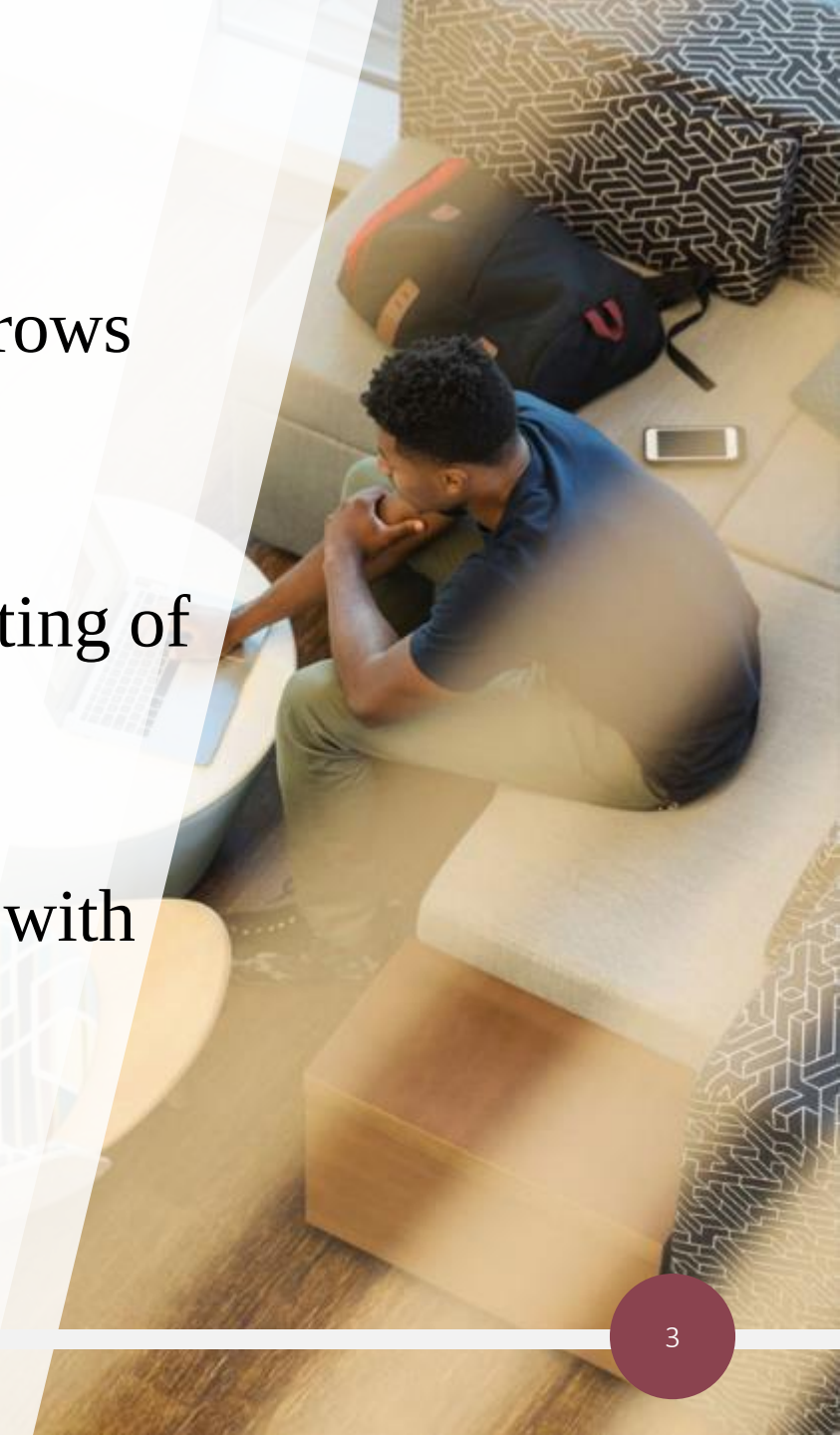
Database Applications with JDBC

- ✓ Connect to and perform database SQL operations, process query results using JDBC API



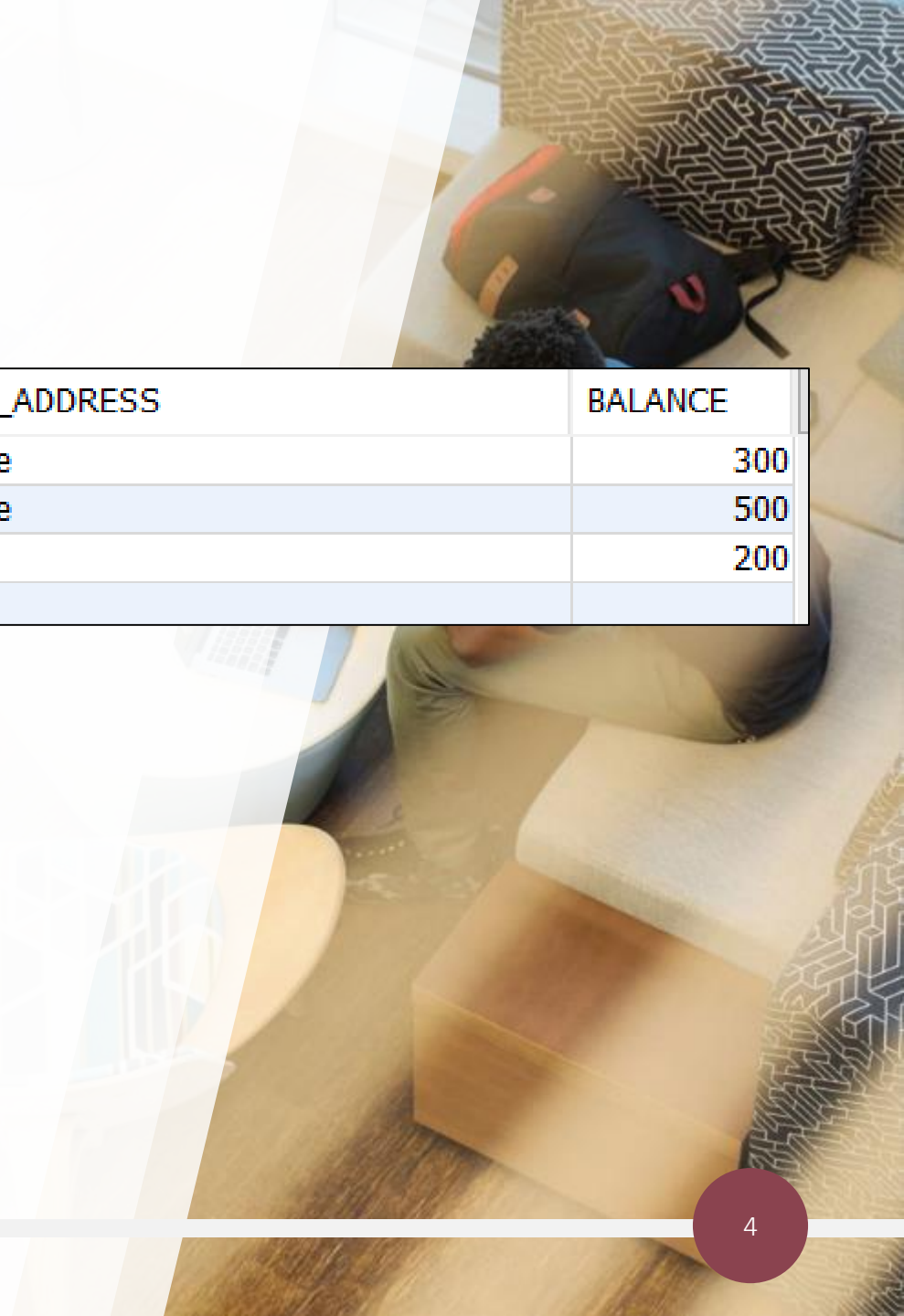
JDBC

- JDBC (Java Database Connectivity) accesses data as rows and columns.
- A relational database organises data into tables consisting of rows and columns.
- We use SQL (Structured Query Language) to interact with a relational database.
- Examples use a Derby database.



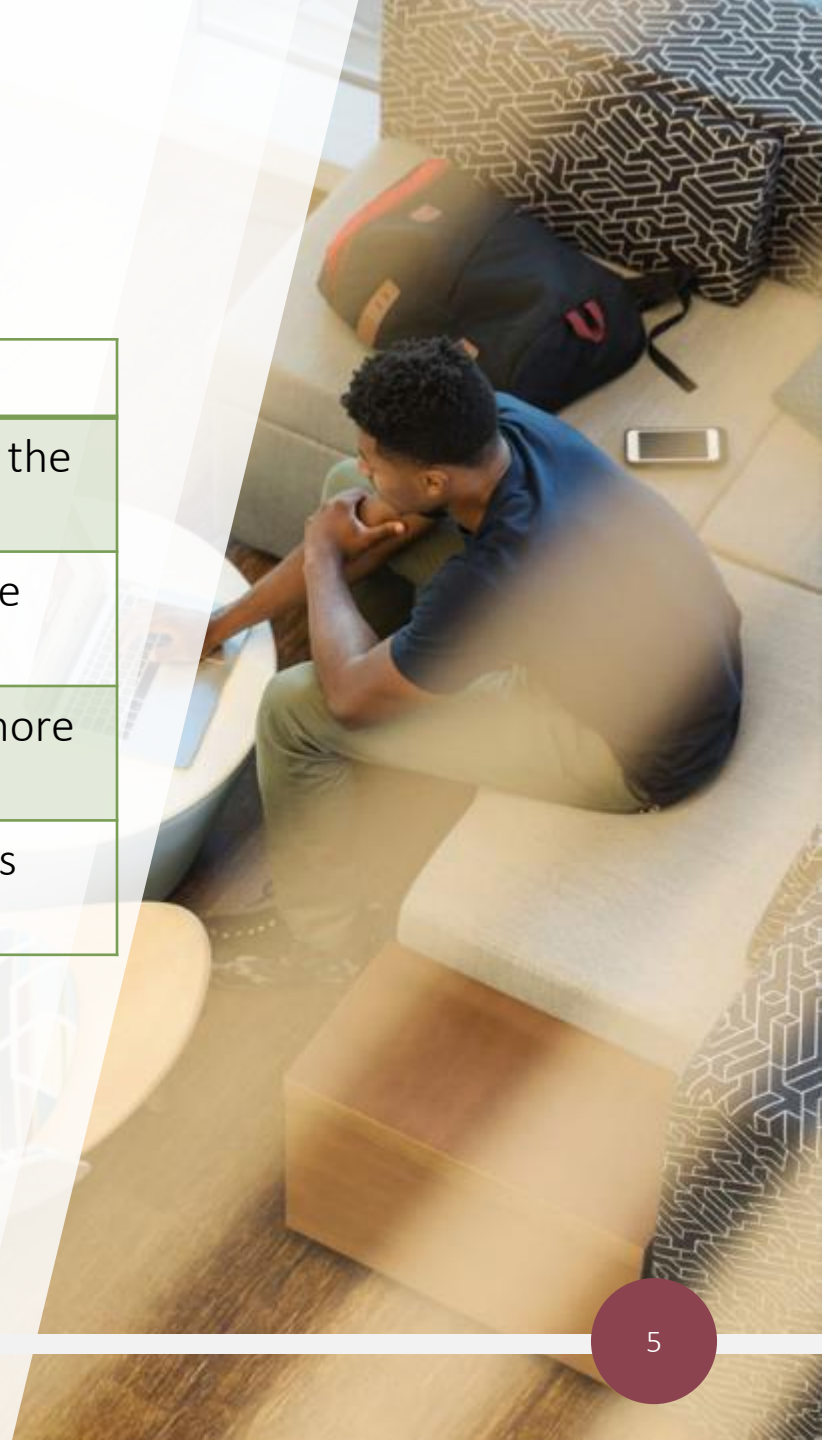
Sample Relational Database

BRANCH_CODE	ACCOUNT_NUMBER	CUST_NAME	CUST_ADDRESS	BALANCE
123456	12345678	Joe Bloggs	Athlone	300
111111	87654321	Ann Bloggs	Athlone	500
222222	67676767	Jane Doe	Dublin	200



CRUD Operations

Operation	SQL keyword	Description
Create	INSERT	Inserts a new row into the table
Read	SELECT	Retrieves data from the table
Update	UPDATE	Changes data in 0 or more rows (in the table)
Delete	DELETE	Deletes 0 or more rows (from the tale)



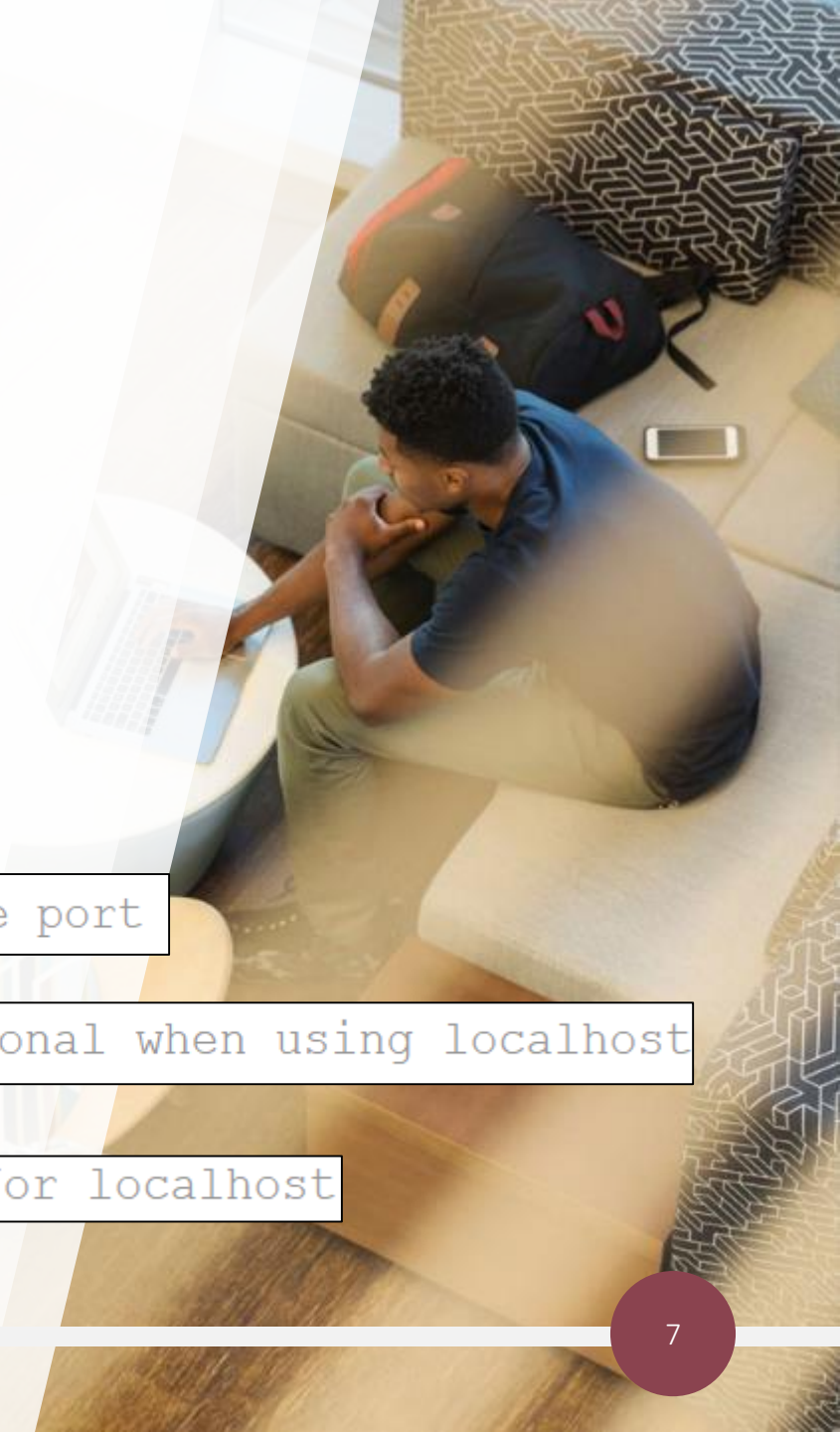
JDBC

- Explain bank_table_SQL.txt

Connecting to a Database

- JDBC URL:
  - protocol : subprotocol : subname
  - protocol is always *jdbc*
  - subprotocol is the vendor or product name – *derby*
    - *mysql*, *oracle*, *postgresql*
  - subname is the database connection details:

```
"jdbc:derby://localhost:1527/BANK_DB"   // including the port
"jdbc:derby://localhost/BANK_DB"        // port is optional when using localhost
"jdbc:derby://127.0.0.1/BANK_DB"        // ip address for localhost
```



```java
package lets_get_certified.jdbc;

import java.sql.DriverManager;      // factory class for creating the db Connection
import java.sql.Connection;         // required interface
import java.sql.PreparedStatement;  // required interface
import java.sql.ResultSet;          // required interface

import java.sql.SQLException;

public class BankService {

    private static Connection con;
    private static BankService bank = new BankService(); // connect to db

    public BankService() {
        try {
            con = DriverManager
                    .getConnection("jdbc:derby://localhost:1527/BANK_DB",
                    "sean", "sean"); // don't expose your password!
            System.out.println("DB connection OK!");
        } catch (SQLException ex) { // SQLEXception is a checked exception
            System.err.println("Exception.");
            ex.printStackTrace();
        }
    }
}
```

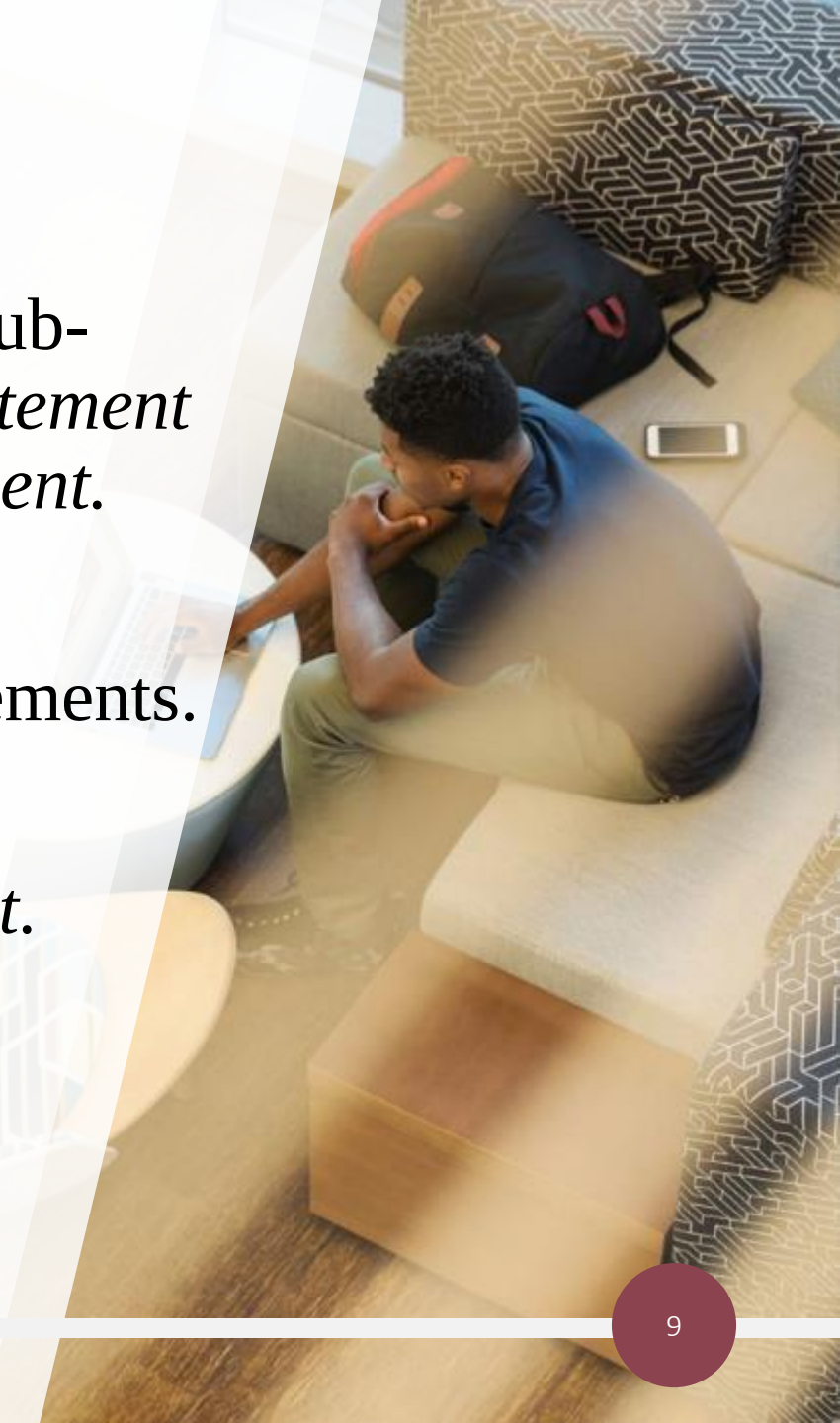
java –cp <path_to_derby>/derby.jar BankService.java

PreparedStatement

- Both *PreparedStatement* and *CallableStatement* are sub-interfaces of *Statement*. We will talk about *CallableStatement* later; for the moment we will focus on *PreparedStatement*.

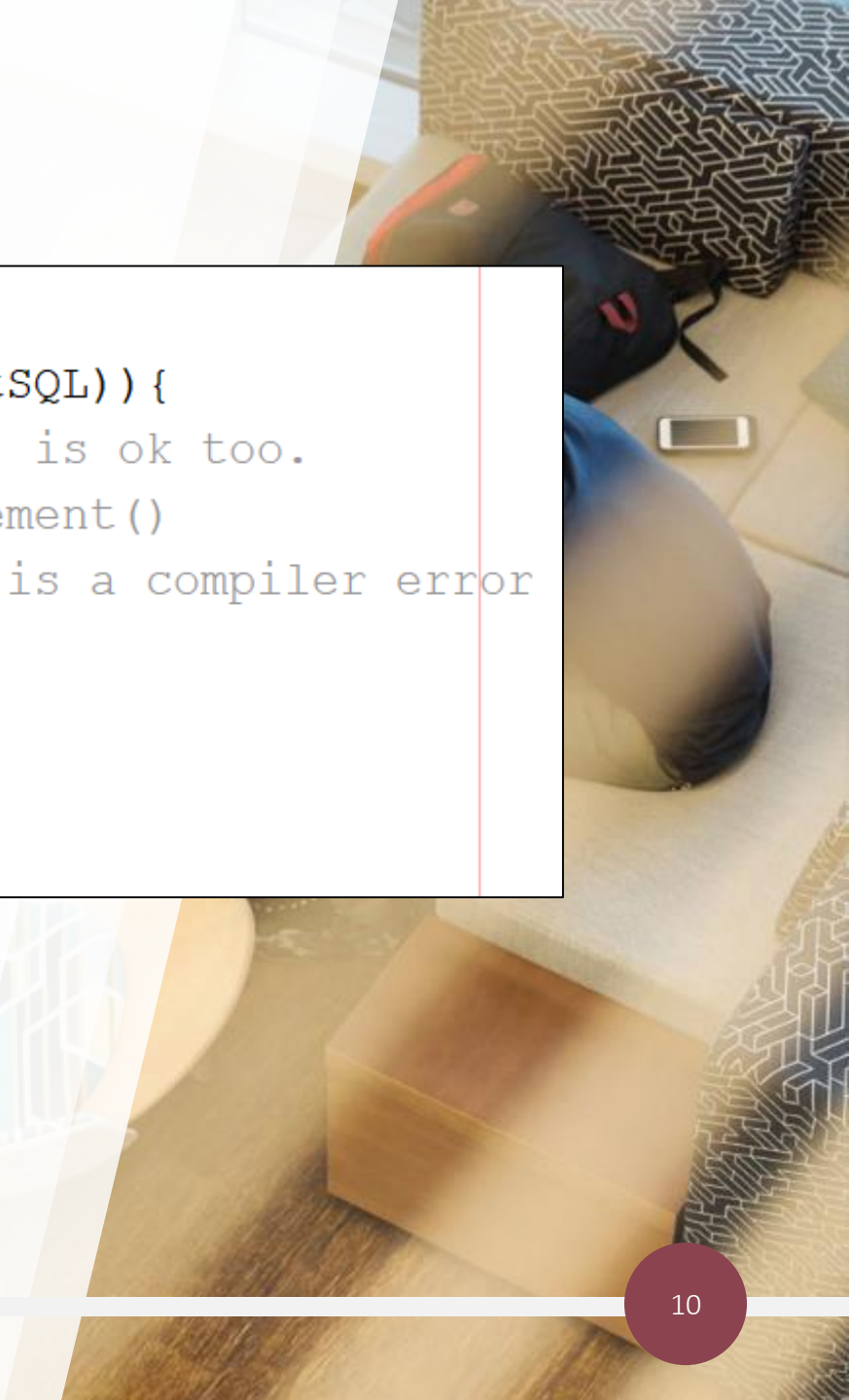
- A *PreparedStatement* enables us to execute SQL statements.

- Use the *Connection* object to get a *PreparedStatement*.
  - *DriverManager* → *Connection*
    - *Connection* → *PreparedStatement*
      - *PreparedStatement* → execute the SQL



PreparedStatement

```
String selectSQL = "SELECT * FROM APP.BANK_TABLE";
try (PreparedStatement ps = con.prepareStatement(selectSQL)){
    // Note: 'var ps = con.prepareStatement(selectSQL)' is ok too.
    // Failure to pass SQL (a String) into prepareStatement()
    // is a compiler error i.e. con.prepareStatement() is a compiler error
    // Do something with 'ps'...
} catch (SQLException sqle) {
    sqle.printStackTrace();
}
```



PreparedStatement

- Once we have the *PreparedStatement* we can now run the SQL.

- The type of SQL statement determines the way you run it.

- SELECT
  - ResultSet rs = ps.**executeQuery();**
- INSERT, UPDATE, DELETE
  - int rowsAffected = ps.**executeUpdate();**
- SELECT, INSERT, UPDATE, DELETE
  - boolean isResultSet = ps.**execute();**
    - true – ResultSet rs = ps.**getResultSet()**; // we ran a query
    - false – int rowsAffected = ps.**getUpdateCount()**; // we ran an ‘update’

JDBC

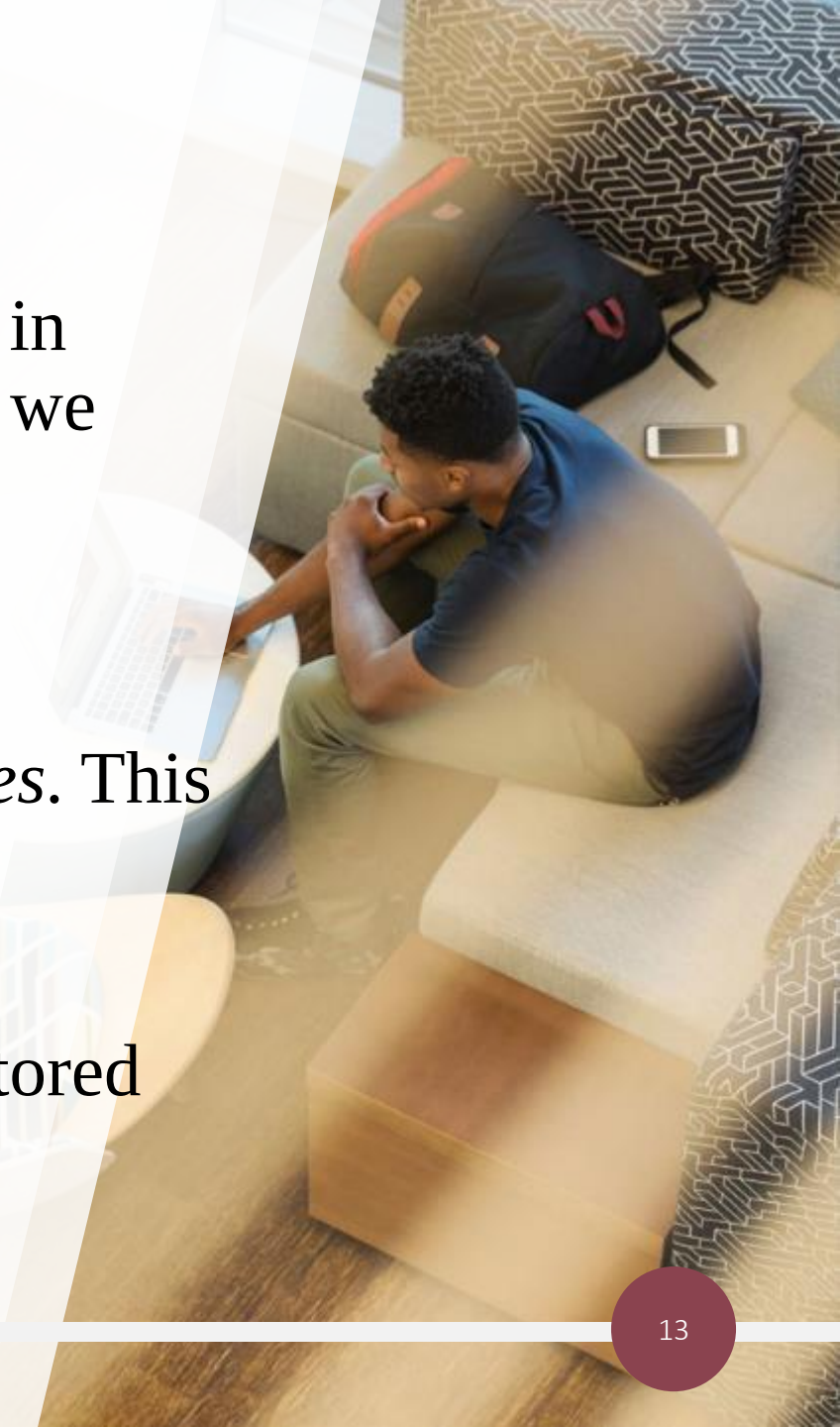
- Bank Example
  - bind variables
  - cursor
  - columns start at 1
  - ResultSet

CallableStatement

- Rather than have your SQL (as a *PreparedStatement*) in your Java code which must be sent across the network; we can have our SQL stored on the database in a “stored procedure”.

- SQL can be stored in the database as *stored procedures*. This reduces network traffic.

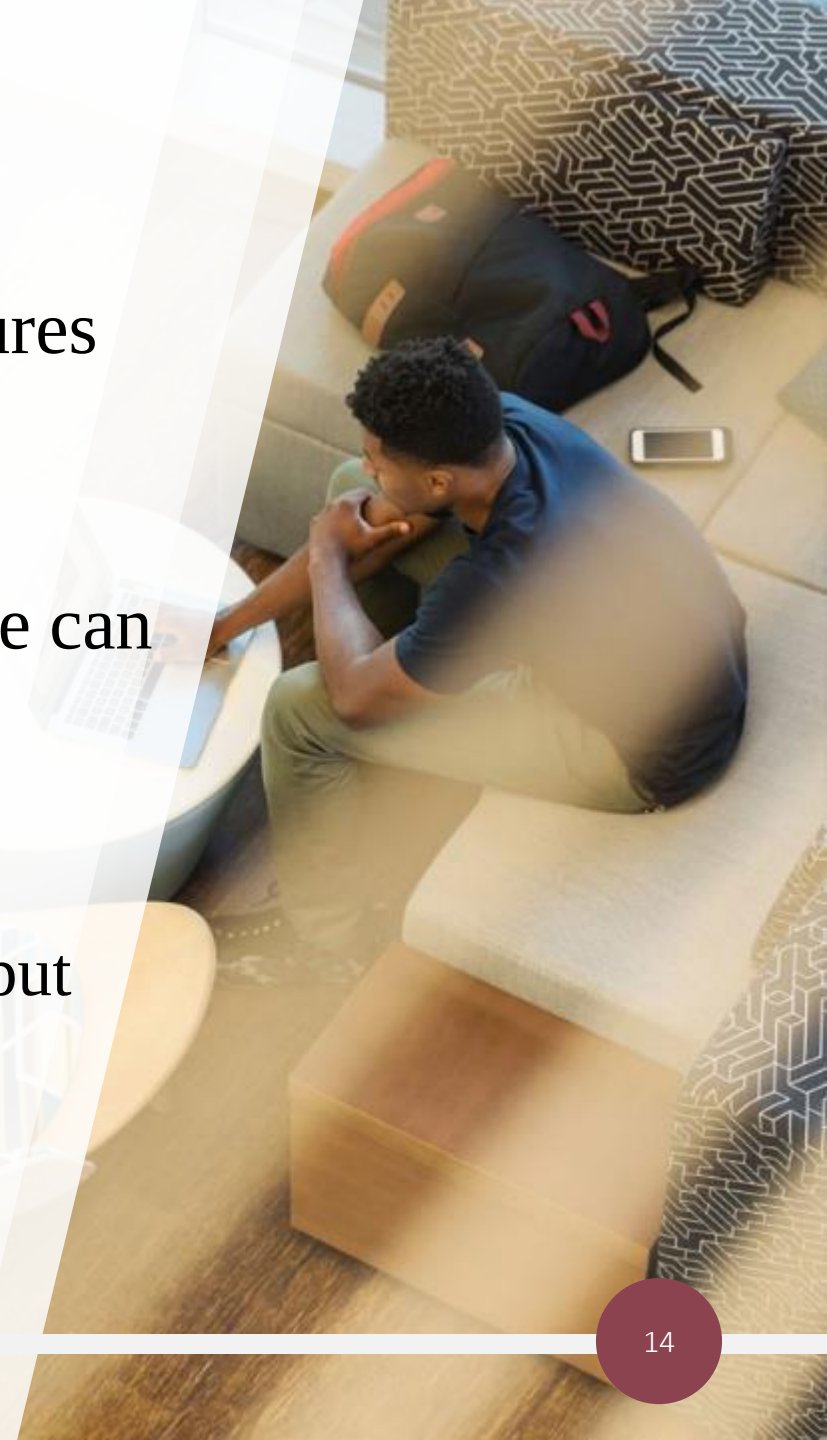
- We use *CallableStatement*’s when we working with stored procedures.



CallableStatement

- We do not need to be able to read/write stored procedures but we must know how to call/execute them.

- In addition to having no parameters, a stored procedure can specify the following parameters:
  - IN – an input parameter
  - OUT – an output parameter
  - INOUT - a parameter that serves for both input and output

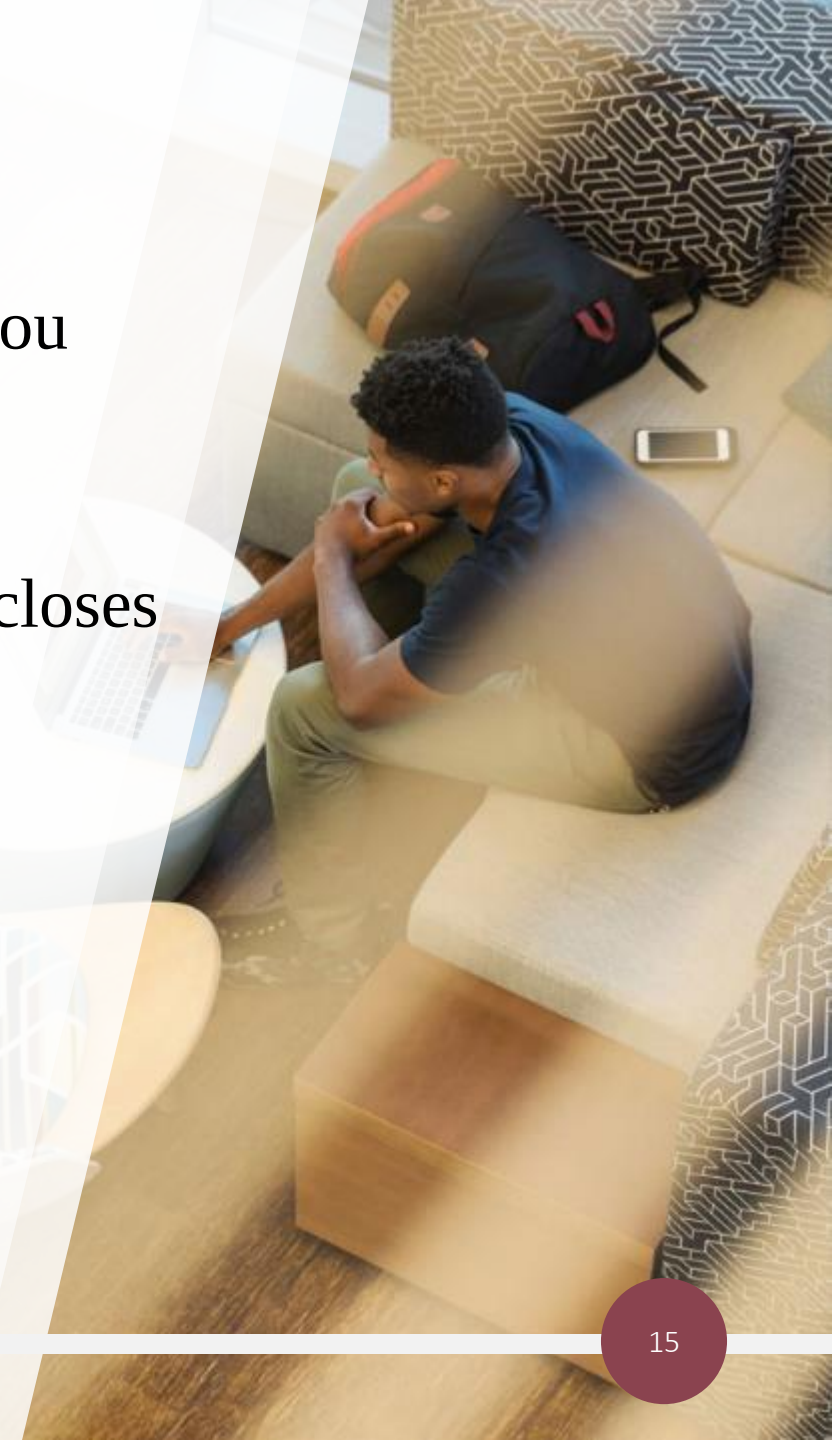


Resource Leaks

- Database resources are expensive to create so make sure you close them when finished.

- *try-with-resources* is very helpful here as it automatically closes resources for us.

- The order is important:
  1. *ResultSet*
  2. *PreparedStatement* or *CallableStatement*
  3. *Connection*



Resource Leaks

- *try-with-resources* closes resources "automatically, in the reverse order from which they were initialized" [JLS]

```
String url = "jdbc:derby://localhost:1527/BANK_DB";
String user = "sean", pwd = "sean";
String sql = "SELECT * FROM APP.BANK_TABLE";
// These resources will be closed automatically in reverse order:
//      ResultSet, PreparedStatement, Connection
// This is the order that we want.
try(Connection con        = DriverManager.getConnection(url, user, pwd);
    PreparedStatement ps = con.prepareStatement(sql);
    ResultSet rs         = ps.executeQuery()){
        // process 'rs'
```