



# Desenvolvimento de Sistemas Software

Aula Teórica 19

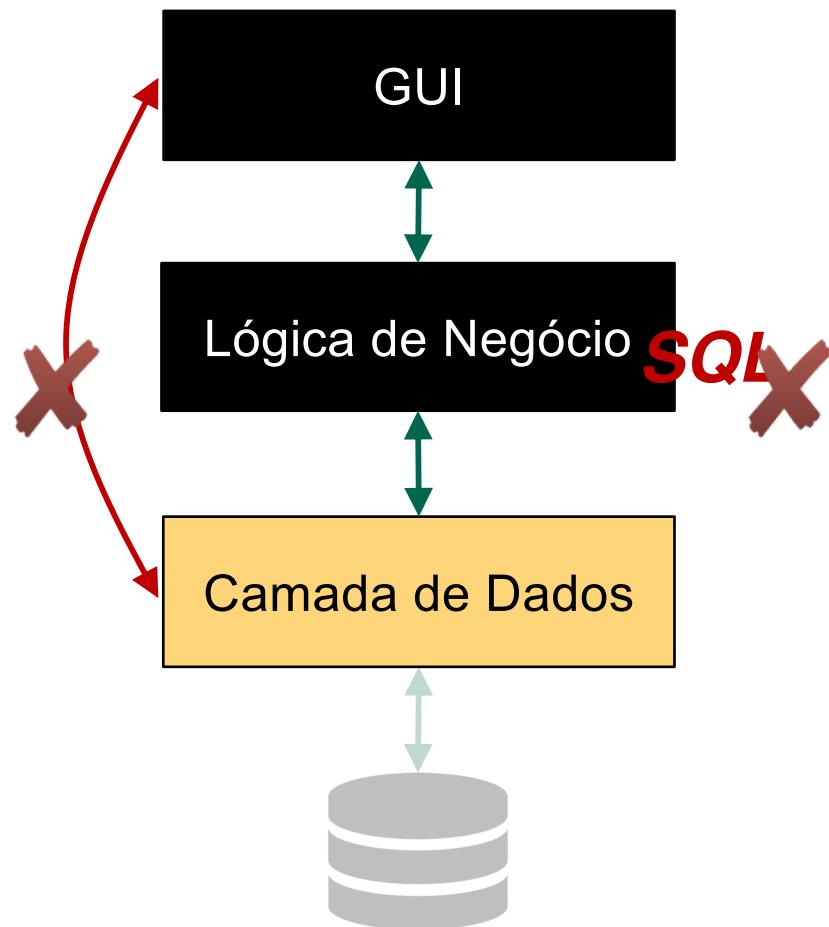
## JDBC - Java DataBase Connectivity

# Contexto

- Bases de dados fornecem persistência.
- Uma correcta organização do código é importante para assegurar e escalabilidade e facilidade de manutenção do código
- A proposta é isolar a ligação à Base de Dados na Camada de Dados.
  - Esta camada permite isolar o acesso aos dados, por forma a que o resto da aplicação não esteja dependente da origem ou estrutura sob a qual os dados estão armazenados.



## Algumas considerações



- Cada camada deve comunicar apenas com as camadas adjacentes
- Uma camada não deve tratar de responsabilidades de outras camadas
- As camadas deverão ter *Facades* para comunicar com outras camadas



# JDBC - Java DataBase Connectivity

- Uma API para acesso a Bases de Dados, para aplicações Java
  - Essencialmente um conjunto de interfaces Java distribuídas pelos pacotes:  
`java.sql` e `javax.sql`
- Implementações de JDBC são disponibilizadas pelos fornecedores de Bases de Dados (e.g. MySQL, PostgreSQL, SQLite, etc.).
  - É necessário carregar o *driver* específico da Base de Dados utilizada.



# JDBC - Java DataBase Connectivity

- JDBC fornece uma API Java que é independente da Base de Dados utilizada.
  - A interação é feita através de classes/interfaces específicas, que são as mesmas independentemente da Bases de Dados.

- Exemplos:

- Abrir uma ligação à Base de Dados (Connection):

```
Connection c = DriverManager.getConnection();
```

- Criar uma *query*:

```
PreparedStatement ps =
c.prepareStatement("SELECT * FROM TABLE WHERE ID < ?")
```



# JDBC - Java DataBase Connectivity

- Misturar objectos do JDBC com objectos da Lógica de Negócio vai originar um **código confuso e de difícil manutenção**
- É necessário encapsular (abstrair) os detalhes da comunicação com a Base de Dados
  - **Data Access Objects (DAOs)**
  - Separação clara entre camada de dados e lógica de negócio

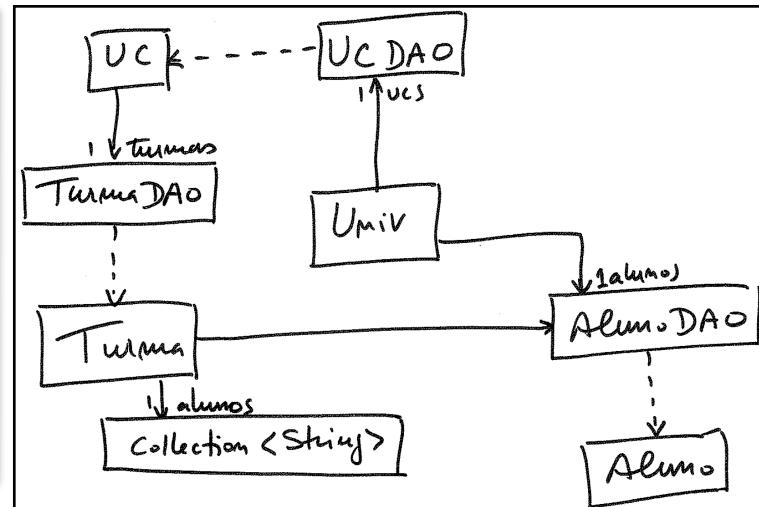
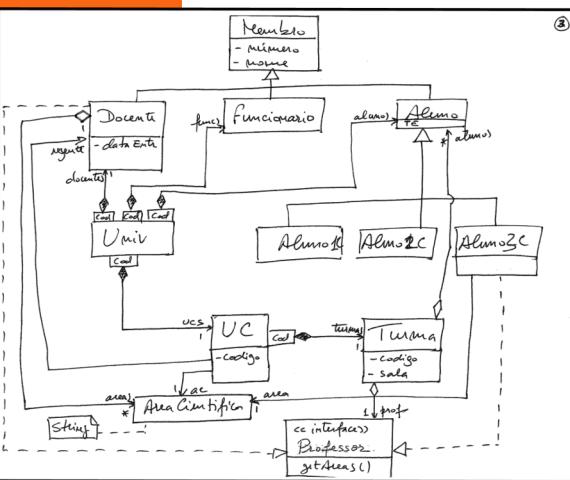
# DAO - Data Access Object

- Padrão para implementar a camada de persistência
- DAOs são classes que:
  - Persistem objectos em Bases de Dados
  - Criam objectos a partir da informação na Base de Dados
  - Encapsulam *queries* SQL
- Podem persistir uma classe ou várias classes
  - Não é obrigatório ter um DAO por classe
- São as *Facades* da Camada de Dados

# DAO - Data Access Object

- Para cada DAO, implementamos um conjunto base de operações base:
  - **Guardar** - `put (k: Object, o: Object) : void`
  - **Procurar** - `get (key: Object) : Object`
  - **Listar** - `list (?) : List<Object>`
  - **Apagar** - `remove (key: Object) : void`
  - **Contar** - `size () : int`
- Temos uma implementação parcial da interface Map
  - Abordagem permite DAOs uniformes

# Código...



```
class Turma {
    private String codigo, sala;
    private Professor prof;
    private Collection<String> alunos;
    private AlunoDAO alDAO;

    void setProf(Professor f) {
        this.prof = f;
    }

    public boolean validaProf() {
        boolean fim = false;
        Iterator<String> alunosIt = alunos.iterator();

        while (!fim & alunosIt.hasNext()) {
            Aluno a = alDAO.get(alunosIt.next());
            fim = a.equals(prof);
        }
        return !fim;
    }
}
```

```
class AlunoDAO implements Map<String,Aluno>{
    ???

    ...
    public Aluno get(Object key) {
        ???
    }
    ...
}
```



# JDBC - Java DataBase Connectivity

- Passos usuais para utilizar JDBC:
  1. Instalar o *driver* JDBC da Base de Dados (uma vez por projecto)  
(e.g. MySQL Connector/J, PostgreSQL JDBC Driver, SQLite JDBC Driver)
  2. Inicializar o *driver* (de cada vez que a aplicação é executada);
  3. Estabelecer uma ligação;
  4. Executar as operações;
  5. (Fechar a ligação)

# JDBC - 1. Instalar o driver

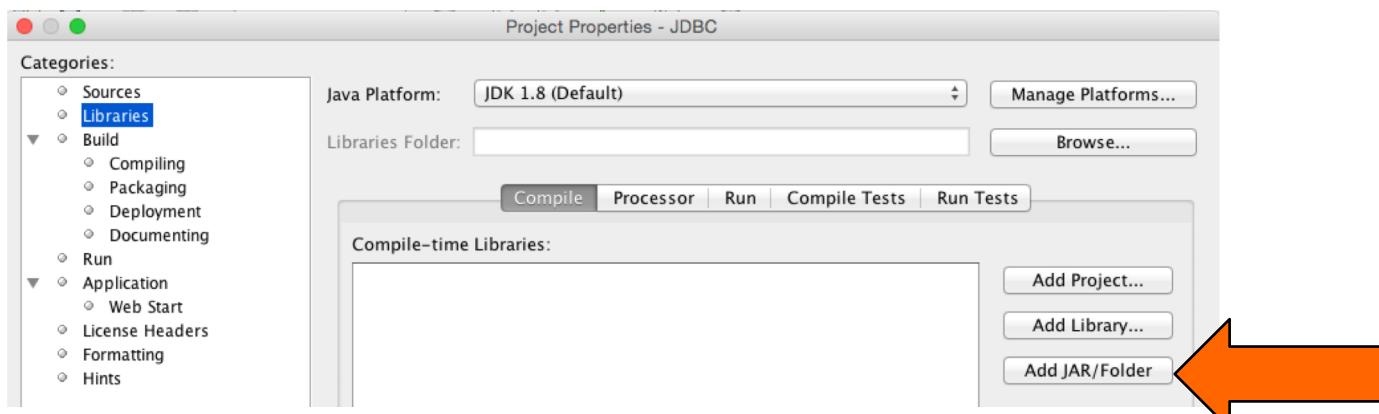
- Fazer download do *driver* (jar) correspondente:
  - Exemplo para MySQL (connector/j):

<http://dev.mysql.com/downloads/connector/j/>



- Importar o *driver* para o projeto (depende do IDE):
  - Exemplo NetBeans:

Project > Properties > Libraries



mysql-connector-jar-5.1.24-bin.jar

## JDBC - 2. Inicializar o driver

- Adicionar *packages* respectivos:

```
import java.sql.*;
```

- Inicializar o *driver* (utiliza-se o método `Class.forName`)
  - exemplo MySQL:

```
Class.forName("com.mysql.jdbc.Driver");
```

**exceção:** `ClassNotFoundException`

```
class AlunoDAO implements Map<String,Aluno> {  
    ???  
  
    public AlunoDAO() {  
        try {  
            Class.forName("com.mysql.jdbc.Driver");  
        }  
        catch (ClassNotFoundException e) {  
            // Driver não disponível  
        }  
    }  
    ...  
}
```

## JDBC - 2. Inicializar o driver

- Outros *drivers*:

- Oracle:

```
try {  
    Class.forName("oracle.jdbc.driver.OracleDriver");  
}  
catch (ClassNotFoundException e) { ... }
```

- PostgreSQL:

```
try {  
    Class.forName("org.postgresql.Driver");  
}  
catch (ClassNotFoundException e) { ... }
```

- SQLite

```
try {  
    Class.forName("org.sqlite.JDBC");  
}  
catch (ClassNotFoundException e) { ... }
```



## JDBC - 3. Estabelecer ligação

- A classe **DriverManager** disponibiliza os seguintes métodos de classe:

```
Connection getConnection(String url);
```

```
Connection getConnection(String url, String login, String pass);
```

```
Connection getConnection(String url, java.util.Properties info);
```

excepção: SQLException

- **Connection** é uma interface que define um conjunto de métodos para realizar operações na base de dados.



## JDBC - 3. Estabelecer ligação: Connection

- A Connection deve ser aberta **antes** de executarmos a operação (e fechada depois)
- As Connection têm um *timeout*
  - Operações realizadas antes de se abrir a conexão, ou após esta ser fechada, geram um excepção.
- Abrir a Connection pode falhar
  - A excepção SQLException tem que ser tratada
- Existe um **número máximo** de Connection permitidas.

## JDBC - 3. Estabelecer ligação

- Ligação por URL:

protocolo:subprotocolo:identificador

- **protocolo**: é constante e representado pela *string* "jdbc"
- **subprotocolo**: é função do motor da base de dados e da forma de acesso (direta ou indireta, por exemplo através de ODBC). No caso de MySQL o subprotocolo é "mysql".
- **identificador**: Indica dados da base de dados a utilizar. No caso do MySQL o identificador é um URI da forma:  
`//<host>/<database>?user=<username>&password=<password>`

## JDBC - 3. Estabelecer ligação

- Exemplo MySQL para ligar a uma Base de Dados local, com nome **alunos**, em que username é **un** e password é **pwd**:

```
class AlunoDAO implements Map<String,Aluno> {  
    private Connection con;  
  
    public AlunoDAO() {  
        try {  
            Class.forName("com.mysql.jdbc.Driver");  
        }  
        catch (SQLException e) { // Driver não disponível }  
  
        public Aluno get(Object key) {  
            try (con =  
                 DriverManager.getConnection("jdbc:mysql://localhost/alunos?user=un&password=pwd")) {  
                ???  
            }  
            catch (SQLException e) {  
                // Erro ao estabelecer a ligação  
            }  
        }  
        ...  
    }  
}
```

## JDBC - 3. Estabelecer ligação

- Outras configurações:

- Oracle:

```
DriverManager.getConnection(  
    "jdbc:oracle:thin:@localhost:1521:database",  
    "username", "password");
```

- PostgreSQL:

```
DriverManager.getConnection(  
    "jdbc:postgresql://localhost:5432/database",  
    "username", "password");
```

- SQLite:

```
DriverManager.getConnection("jdbc:sqlite:test.db");
```

## JDBC - 4. Executar operações SQL

- Três tipos de commandos SQL podem ser executados:

### A. DDL: Data Definition Language

```
CREATE TABLE TAlunos (
    numero VARCHAR(10) NOT NULL,
    nome VARCHAR(50),
    primary key(numero)
```

### B. Selecção - ler dados existentes

```
SELECT * FROM TAlunos
```

### C. Actualização - inserir novos valores na base de dados e/ou modificar os dados existentes

```
UPDATE TAlunos
    SET nome = "João"
    WHERE numero = "123"
```



## JDBC - 4. Executar operações SQL

- A interface **Statement** contém os métodos para realizar operações nos dados.
- Criar um **Statement**:
  - Interface **Connection**: `Statement createStatement();`
  - exceção: `SQLException`

```
public Aluno get(Object key) {  
    try (con =  
        DriverManager.getConnection("jdbc:mysql://localhost/alunos?user=un&password=pwd")) {  
        //create the statement  
        Statement st = con.createStatement();  
        ???  
    }  
    catch (SQLException e) {  
        // Erro ao estabelecer a ligação  
    }  
}
```

## JDBC - 4. Executar operações SQL: Seleção

- Interface Statement

```
ResultSet executeQuery(String sql);
```

excepção: SQLException

- Exemplo para comandos de seleção

```
public Aluno get(Object key) {  
    try (con =  
        DriverManager.getConnection("jdbc:mysql://localhost/alunos?user=un&password=pwd")) {  
        //create the statement  
        Statement st = con.createStatement();  
        String sql = "SELECT * FROM TAlunos WHERE numero= '" + (String)key+ "'";  
        ResultSet rs = st.executeQuery(sql);  
        ???  
    }  
    catch (SQLException e) {  
        // Erro ao estabelecer a ligação  
    }  
}
```

## JDBC - 4. Executar operações SQL: ResultSet

- Comandos de selecção resultam num conjunto de dados.
- Acesso aos resultados é feito através do **ResultSet**.
- Interface ResultSet
  - Funciona como iterador sobre os registos devolvidos

```
boolean next();
```
  - Dentro de um registo fornece um conjunto de métodos para aceder aos campos, por exemplo:

```
getString(int índiceDoCampo);  
getString(String nomeDoCampo);
```
- Um ResultSet está disponível até ser fechado, ou o Statement ser reutilizado ou fechado.

## JDBC - 4. Executar operações SQL: ResultSet

- Exemplo

```
public Aluno get(Object key) {  
    try (con =  
        DriverManager.getConnection("jdbc:mysql://localhost/alunos?user=un&password=pwd")) {  
        //create the statement  
        Statement st = con.createStatement();  
        String sql = "SELECT * FROM TAlunos WHERE numero= '"+(String)key+"'";  
        ResultSet rs = st.executeQuery(sql);  
        Aluno al = null;  
        if (rs.next()) {  
            al = new Aluno(rs.getString(1),rs.getString(2));  
        }  
        return al;  
    }  
    catch (SQLException e) {  
        // Erro ao estabelecer a ligação  
    }  
}
```



## JDBC - 4. Executar operações SQL: Actualização/DDL

- interface Connection:

```
int executeUpdate(String sql);
```

- Se for um comando de atualização (UPDATE, INSERT ou DELETE) Devolve o número de registos afectados
- Comandos DDL (ex: CREATE TABLE) devolvem 0

```
public Aluno put(String key, Aluno value) {  
    try (con =  
        DriverManager.getConnection("jdbc:mysql://localhost/alunos?user=un&password=pwd")) {  
        //create the statement  
        Statement st = con.createStatement();  
        String sql = "DELETE FROM TAlunos WHERE numero='"+key+"'";  
        st.executeUpdate(sql);  
        sql = "INSERT INTO TAlunos VALUES ('"+key+"','"+value.getNome()+"')";  
        st.executeUpdate(sql);  
        return new Aluno(value.getNumero(),value.getNome());  
    }  
    catch (SQLException e) {  
        // Erro ao estabelecer a ligação  
    }  
}
```

## JDBC - 4. Executar operações SQL: SQL Injection

- JDBC não lida com questões de segurança (e.g. *SQL injection*)
- Input do utilizador é sempre (potencialmente) malicioso!
  - st.executeQuery("SELECT \* FROM TAlunos WHERE nome = "+name);
  - E se name for:
    - `` OR '1' = '1'; DELETE \* FROM Talunos;" ?!
  - A este tipo de ataque chama-se **SQL injection**



## JDBC - 4. Executar operações SQL: PreparedStatement

- **PreparedStatement** permitem evitar estes problemas e efectuar *queries* de forma mais eficaz.
- Parâmetros das queries são definidos com ?, e o valor é atribuído com os seguinte métodos:

```
setString(indiceDoParametro, string);  
setInt(indiceDoParametro, inteiro);  
setFloat(indiceDoParametro, float);  
...
```

- Exemplo:

```
PreparedStatement st;  
st = con.prepareStatement("SELECT * FROM TALunos WHERE nome=?");  
st.setString(1, val);  
rs = st.executeQuery();
```

## JDBC - 4. Executar operações SQL: Transaction

- A interface Connection suporta também transacções.
- Permite executar um conjunto de operações, garantindo unicidade das operações.
- É conseguido à custa da configuração da instância de Connection.
- API:

```
con.setAutoCommit(false); //inicia transacção
```

```
con.commit(); //efectua transacção
```

```
con.rollback(); //anula operações da transacção
```

- Excepção:

```
SQLException //no caso de transacção ser abordada
```

## JDBC - 4. Executar operações SQL: Transaction

- Exemplo de transacção:

```
try (con = DriverManager.getConnection("...")) {  
    con.setAutoCommit(false); //inicia transacion  
  
    st = con.prepareStatement("DELETE FROM ...");  
    st.executeUpdate();  
  
    st = con.prepareStatement("INSERT INTO ...");  
    st.executeUpdate();  
  
    con.commit(); //efectua transaction  
} catch (SQLException e) {  
    con.rollback(); //anula transaction  
}
```

## JDBC - 4. Executar operações SQL: Timeout

- Ligações não permanecem activas indefinidamente.
- Se deixarmos uma ligação aberta, eventualmente ela vai expirar.
- Solução:
  - Abrir a conexão antes de efetuar as operações;
  - Fechar antes de retornar.

## JDBC - 5. Fechar ligação

- Interface **Connection** fornece os métodos:

```
void close();  
  
boolean isClosed();  
  
excepção: SQLException
```

- Mas **Connection** implementa `java.lang.AutoCloseable`, pelo que a solução mais simples é utilizar uma expressão ***try-with-resources***:

```
try (con = DriverManager.getConnection("...")) {  
    ...  
} catch (SQLException e) {  
    ...  
}
```

- a **Connection** é fechada automaticamente no fim do **try**

# Sumário

- A comunicação de aplicações Java com bases de dados necessita de:
  - Importar biblioteca para o projecto (através do IDE).
  - Importar no código packages respectivos.
  - Carregar a driver no código.
  - Criar statements e executar.
  - Fechar ligação.
- Transacções devem ser feitas de forma explicita.
- Driver não lida com todos os problemas:
  - É necessário lidar com *timeouts*, *sql injection*, etc.