

JAVA FUNDAMENTOS

# ACESSO AO **BANCO DE DADOS**

THIAGO T. I. YAMAMOTO



**LISTA DE FIGURAS**

Figura 7.1 – Exemplo tabelas livro e editora .....	7
Figura 7.2 – Estrutura JDBC .....	9
Figura 7.3 – Papel do Driver Manager.....	10
Figura 7.4 – Classes e Interfaces do arquivo rt.jar .....	11
Figura 7.5 – Criando uma tabela .....	13
Figura 7.6 – Criando uma tabela - Parte 2 .....	13
Figura 7.7 – Tabela TAB_COLABORADOR.....	14
Figura 7.8 – Criando uma sequência .....	14
Figura 7.9 – Criando uma sequência – Parte 2 .....	15
Figura 7.10 – Botão para executar os comandos SQL.....	16
Figura 7.11 – Resultado da execução do comando select.....	17
Figura 7.12 – Resultado da execução.....	17
Figura 7.13 – Resultado da alteração do registro na tabela .....	20
Figura 7.14 – Atualização de um registro da tabela TAB_COLABORADOR.....	20
Figura 7.15 – Criando um novo projeto Java.....	21
Figura 7.16 – Criando uma pasta para o driver do Oracle – Parte 1 .....	22
Figura 7.17 – Criando uma pasta para o driver do Oracle – Parte 2 .....	22
Figura 7.18 – Adicionando o driver do Oracle no projeto .....	23
Figura 7.19 – Processo transacional .....	48
Figura 7.20 – Exemplo de transação atômica. ....	49
Figura 7.21 – Camada de acesso a dados da aplicação (DAO).....	52

**LISTA DE QUADROS**

Quadro 7.1 – Classe e Interfaces do JDBC.....	10
Quadro 7.2 – Operadores lógicos .....	19
Quadro 7.3 – JDBC Drivers e URL String. ....	26

EMPRE

## LISTA DE TABELAS

Tabela 7.1 – TABELA WHERE CONDICA01 .....	18
---	----

EMSE

## SUMÁRIO

7 ACESSO AO BANCO DE DADOS .....	6
7.1 Introdução .....	6
7.1 Java Database Connectivity .....	8
7.2 Banco de dados oracle e comandos sql .....	12
7.2.1 Cadastrando informações na tabela .....	15
7.2.2 Leitura de dados de uma tabela .....	17
7.2.3 Atualizando valores na tabela .....	20
7.2.4 Remoção de registros de uma tabela (Delete) .....	21
7.2.5 Conectando a base de dados .....	21
7.2.6 Statements .....	26
7.2.7 CallableStatement .....	31
7.3 Controle transacional .....	48
7.3.1 Transação .....	48
7.4 <i>Design patterns</i> .....	51
7.4.1 Data Access Object (DAO) .....	52
7.4.2 DAO Factory .....	65
7.4.3 Abstract Factory .....	68
7.4.4 Singleton .....	71
REFERÊNCIAS .....	74

## 7 ACESSO AO BANCO DE DADOS

### 7.1 Introdução

A maioria dos sistemas precisa armazenar dados e informações para serem recuperados posteriormente. Por exemplo, um sistema bancário precisa registrar várias informações, entre eles os saques e depósitos para manter o saldo da conta atualizada. Um sistema de vendas online deve armazenar todos os dados de seus clientes, produtos, fornecedores, vendas etc.

Uma solução é armazenar as informações em arquivos textos e planilhas. Porém, por se tratar de uma grande quantidade de informações, é difícil de manter o gerenciamento dos dados, e com certeza qualquer acesso a essas informações será com pouca performance e utilizando bastante processamento. Outra opção é utilizar sistemas especializados no armazenamento de dados que oferecem mais funcionalidades e recursos avançados, como backup e buscas mais eficientes. Esses sistemas são conhecidos como: Sistemas Gerenciadores de Banco de Dados – SGBD.

Um Sistema de Gerenciamento de Banco de Dados é um conjunto de programas de computador responsáveis pelo gerenciamento de uma base de dados. Seu principal objetivo é retirar da aplicação cliente a responsabilidade de gerenciar o acesso, manipulação e a organização dos dados. O SGBD disponibiliza uma interface para que seus clientes possam realizar as operações de inclusão, alteração, exclusão e consulta de dados.

Os principais SGBDs utilizados nas empresas aplicam o Modelo Relacional para armazenar informações, e a linguagem SQL é utilizada para realizar as operações nas bases de dados.

Os principais SGBDs utilizados no mercado são:

- Oracle Database.
- Microsoft SQL Server.
- PostgreSQL.
- Sysbase.

- MySQL Server.
- Apache Derby.

O Modelo Relacional é um modelo de dados baseados em entidades e relacionamentos. Uma tabela armazena as informações relevantes da entidade. A atribuição de valores de uma entidade constrói um registro da tabela. A relação determina como cada registro da tabela se associa a registros de outras tabelas. Uma tabela é formada por registros (linhas) e os registros são formados por campos (colunas).

A figura abaixo exemplifica duas tabelas: Livro e Editora.

ISBN	NM_LIVRO	NR_PAGINA	COD_EDITORA
321212211	Aprenda Java	500	1
325654212	SQL Básico	150	2
321545666	Banco de Dados	100	1
COD_EDITORA		NM_EDITORA	
1		Editora FIAP	
2		Editora Nova	

Figura 7.1 – Exemplo tabelas livro e editora  
Fonte: Fiap (2016)

A tabela TAB\_LIVRO armazena as informações: ISBN, nome, número de páginas e código da editora. Ela possui três registros (linhas): Aprenda Java, SQL Básico e Banco de Dados. Já a tabela TAB\_EDITORA armazena somente o código da editora e o nome. E possui somente dois registros, Editora FIAP e Editora Nova. Perceba que a coluna COD\_EDITORA da tabela TAB\_LIVRO armazena o relacionamento dos livros com a editora. Por exemplo, o livro Aprenda Java possui a editora de código 1, ou seja, a Editora FIAP.

As tabelas normalmente têm uma chave primária, que é um identificador único que garante que nenhum registro será duplicado. No nosso exemplo, a coluna ISBN da tabela TAB\_LIVRO e o COD\_EDITORA da tabela TAB\_EDITORA são as colunas que armazenam a chave primária de suas respectivas tabelas.

Structured Query Language (SQL) ou Linguagem de Consulta Estruturada é uma linguagem internacional de pesquisa declarativa padrão para banco de dados relacional, através dela os principais SGBDs interagem com os bancos de dados.

Alguns dos principais comandos SQL para a manipulação de dados são: INSERT (inserção), UPDATE (atualização), SELECT (consulta), DELETE (exclusão). O SQL possibilita também criar tabelas, relações, controle de acesso etc.

Os sistemas desenvolvidos na plataforma Java comunicam-se com o SGBD e manipulam seus dados utilizando a API Java DataBase Connectivity (JDBC).

### **7.1 Java Database Connectivity**

O Java Database Connectivity é uma API (Application Programming Interface), um conjunto de regras que permite uma padronização no acesso aos diversos SGDBs disponíveis no mercado. Para evitar que cada banco tenha a sua própria API, o Java Database Connectivity (JDBC) revela um conjunto de interfaces bem definidas que devem ser implementadas pelas empresas fornecedoras de SGDB e que deseja ser compatível à plataforma Java.

Dessa forma, são os SGDBs que se adaptam à plataforma Java, através da implementação de um JDBC, e não o contrário. Com essa padronização, o desenvolvedor não precisa se preocupar caso precise alterar um sistema para suportar um novo Banco de Dados (SGDB), como existe essa padronização, basta trocar as bibliotecas de acesso de um banco de dados pelo outro. Todo o código implementado para sistema será o mesmo, pois as duas bibliotecas de acesso aos bancos de dados seguem os mesmos padrões.

Essas bibliotecas de classes que permitem a integração de um SGDB à aplicação Java são chamadas de Driver. Geralmente, as empresas fornecedoras de SGDBs oferecem o driver de conexão que seguem às especificações JDBC.

A figura abaixo apresenta a estrutura JDBC:



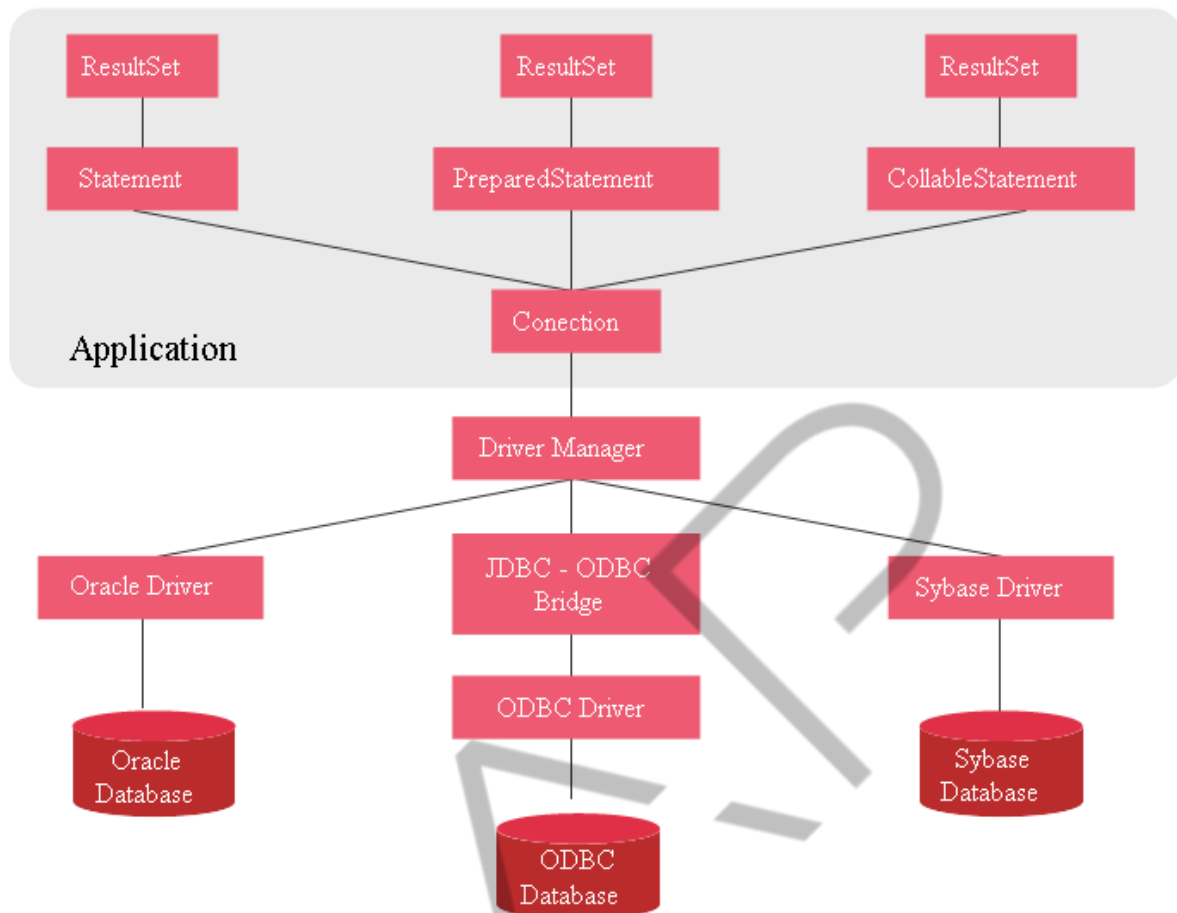


Figura 7.2 – Estrutura JDBC  
Fonte: Fiap (2016)

Na figura é possível observar que a estrutura de classes é a mesma, independentemente do SGDB que será utilizado (Oracle, Microsoft, Sybase). Nesta estrutura, uma das poucas classes que é implementada é a `DriverManager`, responsável por identificar o conjunto de bibliotecas (driver) que será utilizada: `Oracle Driver`, `JDBC-ODBC Driver`, `Sybase Driver`.

No quadro abaixo são apresentadas as principais interfaces e a classe principal que são utilizadas para acesso ao banco de dados através dos padrões JDBC:

Componente	Descrição
DriverManager	Responsável por encontrar o drive e estabelecer a conexão com o SGBDR
Connection	Representa a conexão com o SGBDR por onde serão passados os comandos SQL
Statement	Execução do comando SQL
PreparedStatement	
CallableStatement	
ResultSet	Representa os registros retornados de um Statement, PreparedStatement ou CallableStatement

Quadro 7.1 – Classe e Interfaces do JDBC  
Fonte: Fiap (2016)

As classes para a manipulação de um Banco de Dados estão no pacote **java.sql**. Para se conectar a um Banco de Dados (SGDB) é preciso solicitar a abertura de uma conexão com o Banco de Dados utilizando o Driver Manager, o gerenciador de drivers. Caso o Driver Manager consiga estabelecer uma conexão com o SGDB, um objeto do tipo `java.sql.Connection` é retornado, caso contrário, uma exceção é gerada.

O Driver Manager é responsável por encontrar o driver que será utilizado na aplicação. Quando uma implementação (driver) é carregada, ela é registrada utilizando o Driver Manager. A partir do Driver Manager serão criadas as conexões para a base de dados utilizando o método `getConnection()` que recebe uma String que identifica qual banco de dados vamos conectar.

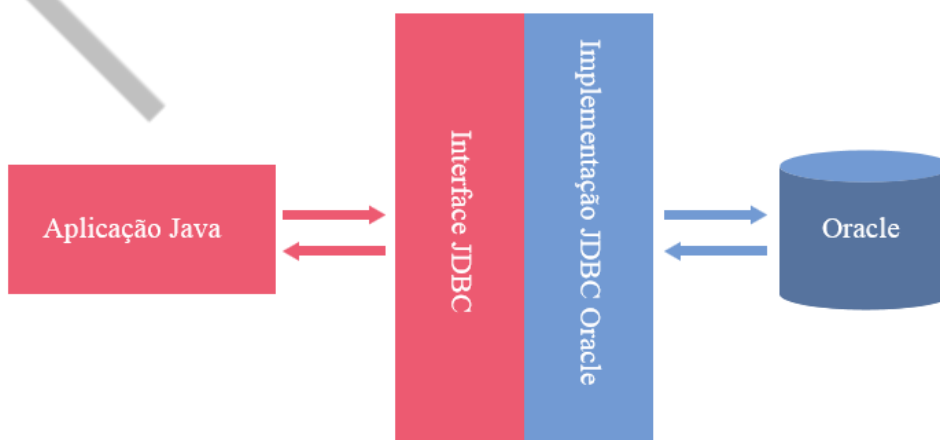


Figura 7.3 – Papel do Driver Manager  
Fonte: Fiap (2016)

Após a conexão, é possível utilizar os objetos do tipo `java.sql.Statement`, `java.sql.PreparedStatement`, `java.sql.CallableStatement` para executar comandos SQL no SGDB, que veremos adiante.

Todas as interfaces e classes da estrutura do JDBC podem ser encontradas no arquivo `rt.jar`, contido no pacote JDK do Java:

















 <code>Blob.class</code>	1/1/1980 00:00	516	0%	516	<code>java\sql\</code>
 <code>CallableStatement.class</code>	1/1/1980 00:00	4.449	0%	4.449	<code>java\sql\</code>
 <code>Clob.class</code>	1/1/1980 00:00	704	0%	704	<code>java\sql\</code>
 <code>Connection.class</code>	1/1/1980 00:00	2.320	0%	2.320	<code>java\sql\</code>
 <code>DatabaseMetaData.class</code>	1/1/1980 00:00	9.882	0%	9.882	<code>java\sql\</code>
 <code>DataTruncation.class</code>	1/1/1980 00:00	1.279	0%	1.279	<code>java\sql\</code>
 <code>Date.class</code>	1/1/1980 00:00	1.818	0%	1.818	<code>java\sql\</code>
 <code>Driver.class</code>	1/1/1980 00:00	490	0%	490	<code>java\sql\</code>
 <code>DriverInfo.class</code>	1/1/1980 00:00	667	0%	667	<code>java\sql\</code>
 <code>DriverManager.class</code>	1/1/1980 00:00	6.580	0%	6.580	<code>java\sql\</code>
 <code>DriverPropertyInfo.class</code>	1/1/1980 00:00	506	0%	506	<code>java\sql\</code>
 <code>ParameterMetaData.class</code>	1/1/1980 00:00	824	0%	824	<code>java\sql\</code>
 <code>PreparedStatement.class</code>	1/1/1980 00:00	1.974	0%	1.974	<code>java\sql\</code>
 <code>Ref.class</code>	1/1/1980 00:00	358	0%	358	<code>java\sql\</code>
 <code>ResultSet.class</code>	1/1/1980 00:00	7.052	0%	7.052	<code>java\sql\</code>
 <code>ResultSetMetaData.class</code>	1/1/1980 00:00	1.067	0%	1.067	<code>java\sql\</code>

Figura 7.4 – Classes e Interfaces do arquivo `rt.jar`  
Fonte: Fiap (2016)

Desenvolver sistemas que acessam banco de dados segue alguns passos em comum, independentemente da linguagem de programação. Podemos fazer uma analogia para entender esses passos: para realizar um telefonema é preciso primeiramente ter o número de telefone da pessoa com a qual queremos conversar. Assim, podemos discar o número em um aparelho telefônico. Se o destinatário estiver disponível, ele atenderá a ligação e assim a conexão estará feita. Depois disso, é possível transmitir informações. No final, vamos desligar a ligação, finalizando a conexão.

Para a comunicação de um sistema a um banco de dados, o processo é o mesmo: primeiro, é necessário ter o “número” do banco de dados destinatário, considerando o endereço de IP (Internet Protocol) como endereço físico e a porta como endereço lógico. Com o banco disponível, vamos obter uma conexão para realizar a comunicação. Através da conexão é possível enviar comandos SQL (Structured Query Language), que serão executados no banco de dados. Por fim, a conexão é fechada, encerrando-se o processo.

Outro ponto importante no processo da comunicação com o banco de dados é o tratamento dos possíveis problemas que podem ocorrer. Assim como em uma ligação, problemas de conexão e comunicação podem acontecer, a diferença é que

na ligação existe a ação direta do usuário, já na programação o próprio programa deve prever e tratar as possíveis situações de erro.

Agora que sabemos os passos para a conexão e comunicação com o banco de dados, vamos criar uma base de dados e uma tabela. Para isso, vamos utilizar o banco de dados Oracle e a ferramenta Oracle Developer.

## 7.2 Banco de dados oracle e comandos sql

Vamos começar com o banco de dados. Primeiro vamos nos conectar à base de dados Oracle utilizando a ferramenta Oracle Developer.

Conecte a base de dados utilizando as seguintes configurações. Depois de conectado, é o momento de criar a primeira tabela para armazenar os dados dos colaboradores de uma empresa, a qual deverá conter as seguintes colunas:

- **CODIGO\_COLABORADOR:** chave primária da tabela, campo obrigatório e com valores únicos.
- **NOME:** nome do colaborador.
- **EMAIL:** e-mail do funcionário.
- **SALARIO:** valor do salário do colaborador.
- **DATA\_CONTRATACAO:** data de contratação.

Para isso, podemos criar a tabela utilizando os comandos SQL (*CREATE TABLE*) ou realizá-la de forma visual, através do Oracle Developer.

Vamos utilizar a forma visual. Primeiro, procure a área de conexões no Oracle Developer, do lado esquerdo, clique com o botão direito do mouse em cima de “Tabelas” e escolha a opção “Nova Tabela”, conforme a figura abaixo:

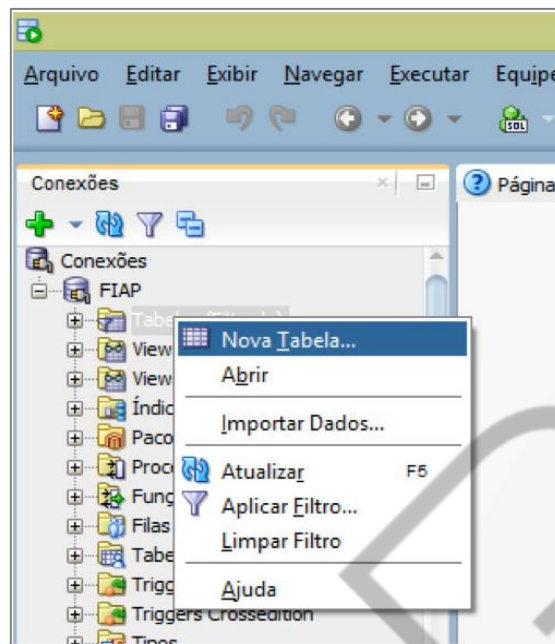


Figura 7.5 – Criando uma tabela  
Fonte: Fiap (2016)

Agora, configure a nova tabela com o nome **TAB\_COLADORADOR** e adicione as colunas, conforme a figura abaixo:

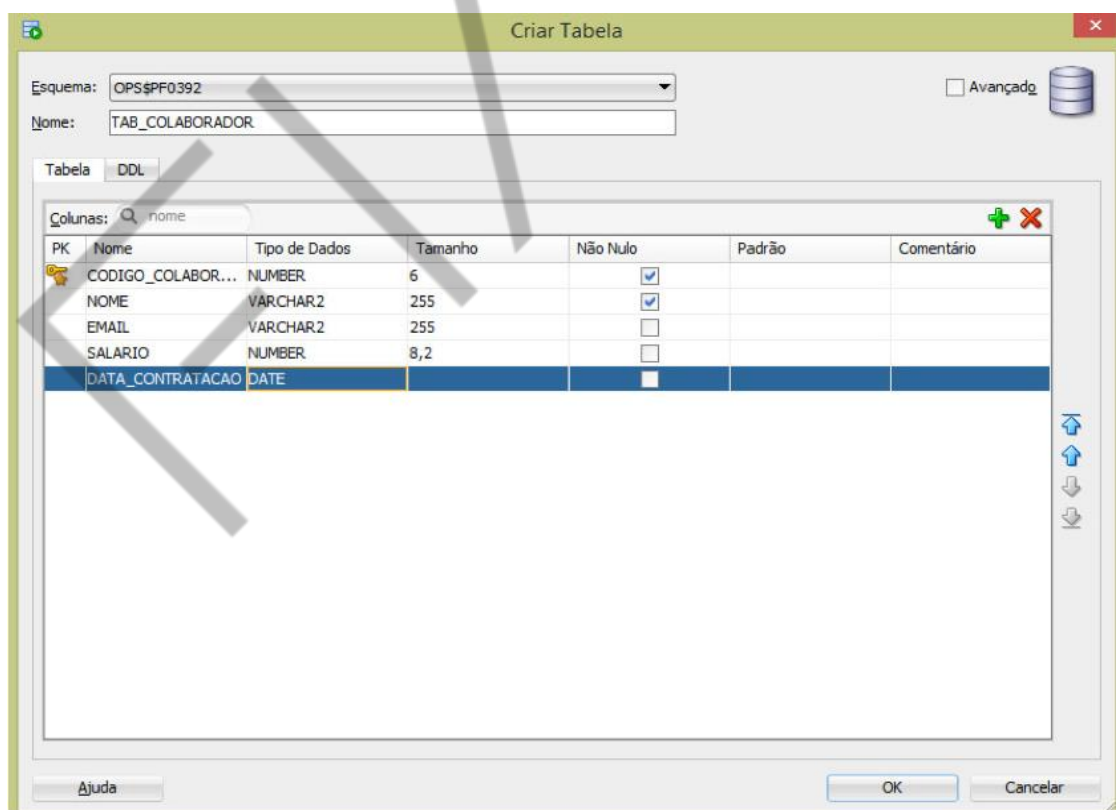


Figura 7.6 – Criando uma tabela - Parte 2  
Fonte: Fiap (2016)

Configure o nome e o tipo de dado de cada coluna. Observe que a coluna **CODIGO\_COLABORADOR** está marcada como chave primária e a coluna **NOME** está marcada como obrigatória. Foram definidos também os tamanhos para cada uma das colunas, menos a coluna de data. Finalize clicando no botão OK.

Dentro da área de conexão, na área de tabelas, procure pela tabela **TAB\_COLABORADOR**, e dê dois cliques para visualizá-la:

COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1 CODIGO_COLABORADOR	NUMBER (6, 0)	No	(null)	1	(null)
2 NOME	VARCHAR2 (255 BYTE)	No	(null)	2	(null)
3 EMAIL	VARCHAR2 (255 BYTE)	Yes	(null)	3	(null)
4 SALARIO	NUMBER (8, 2)	Yes	(null)	4	(null)
5 DATA_CONTRATACAO	DATE	Yes	(null)	5	(null)

Figura 7.7 – Tabela TAB\_COLABORADOR  
Fonte: Fiap (2016)

Pronto! A tabela está criada! Agora vamos criar uma *sequence* para gerar os valores do código do colaborador. Para isso, vá novamente à área de conexões e procure por “Sequências”, clique com o botão direito do mouse e escolha a opção “Nova Sequência”, conforme a figura abaixo:

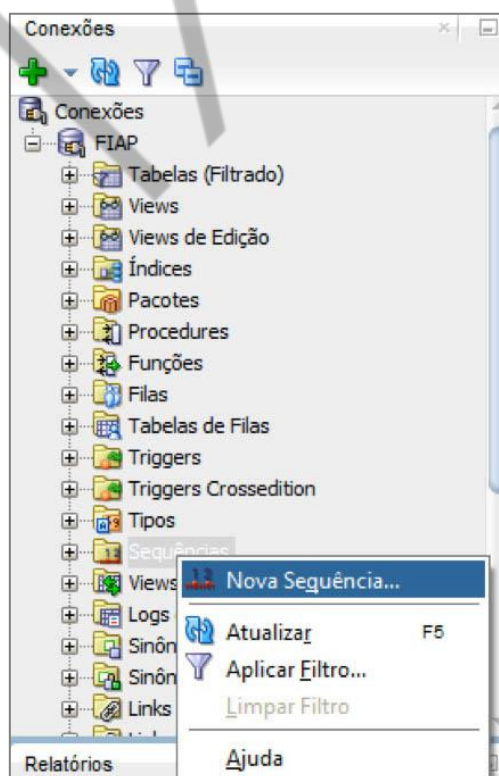


Figura 7.8 – Criando uma sequência  
Fonte: Fiap (2016)

Configure o nome da sequência como SQ\_COLABORADOR e nas propriedades faça a sequência começar com 1 e incrementar de 1 em 1.

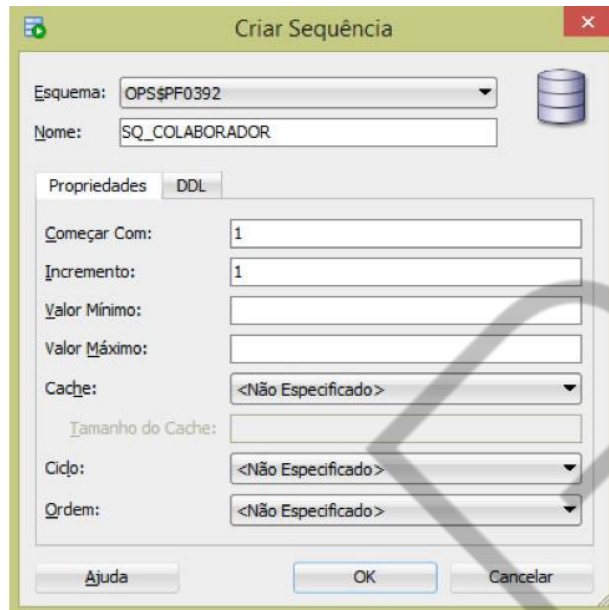


Figura 7.9 – Criando uma sequência – Parte 2  
Fonte: Fiap (2016)

Agora sim, estamos prontos para começar! O próximo passo é conhecer alguns comandos básicos para a manipulação de informações da tabela.

As operações básicas para manipular os dados de uma tabela são: inserir, consultar, alterar e remover. Essas operações são realizadas através de uma linguagem denominada **SQL** (*Structured Query Language*). Essa linguagem oferece quatro comandos básicos que formam o **CRUD**: Create (inserir), Read (consultar), Update (alterar) e Delete (remover).

### 7.2.1 Cadastrando informações na tabela

Através do comando INSERT é possível cadastrar um registro em uma tabela. Podemos utilizá-la de duas formas:

A primeira é informando somente os valores para as colunas, assim, a ordem dos valores será inserida na ordem das colunas na tabela:

**INSERT INTO TABELA VALUES (VALOR1, VALOR2, VALOR3, ...);**

A segunda é informando a coluna e os valores que serão cadastrados:

**INSERT INTO TABELA (COLUNA1, COLUNA2, COLUNA3, ...) VALUES (VALOR1, VALOR2, VALOR3, ...);**

O resultado final é o mesmo, o mais indicado é utilizar a segunda opção, pois caso a estrutura da tabela seja alterada, os valores ainda serão inseridos nas colunas corretas.

**INSERT INTO TAB\_COLABORADOR (CODIGO\_COLABORADOR, NOME, EMAIL, SALARIO, DATA\_CONTRATACAO) VALUES (SQ\_COLABORADOR.NEXTVAL, 'Thiago', 'thiagoyama@gmail.com', 1500, TO\_DATE ('10/10;2010', 'dd/mm/yyyy'));**

O código abaixo apresenta um exemplo de cadastro de um registro na tabela TAB\_COLABORADOR:

Note que utilizamos a sequência SQ\_COLABORADOR para gerar um novo código para o registro. (SQ\_COLABORADOR.NEXTVAL).

Outro detalhe é a função TO\_DATE, que converte um texto para uma data: TO\_DATE('10/10/2010','dd/mm/yyyy'). O primeiro valor é a String que será convertida para data e o segundo é o formato (máscara) da data que foi informada:

- DD – Day (dois dígitos para o dia).
- MM – Month (dois dígitos para mês).
- YYYY – Year (quatro dígitos para o ano).

Para executar um comando SQL no Oracle SQL Developer, utilize o botão execute ou o atalho CTR + ENTER:

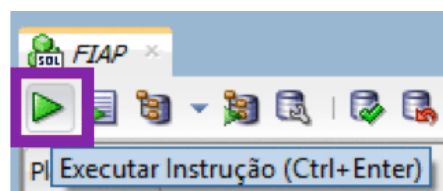


Figura 7.10 – Botão para executar os comandos SQL  
Fonte: Fiap (2016)



### 7.2.2 Leitura de dados de uma tabela

Para recuperar os registros de uma ou mais tabelas utilizamos o comando **SELECT**. A sua sintaxe básica é:

**SELECT COLUNA1, COLUNA2 FROM TABELA;**

A primeira parte da instrução especifica as colunas que queremos recuperar (SELECT COLUNA1, COLUNA2), a segunda parte define a tabela (FROM TABELA). Dessa forma, esse comando recupera as colunas especificadas de todos os registros armazenados na tabela.

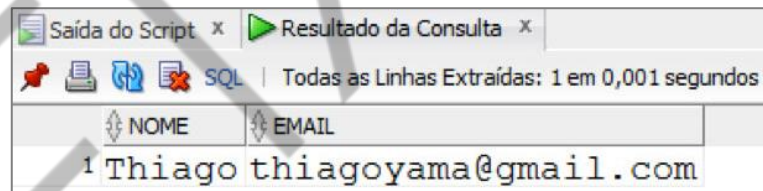
É possível utilizar o caractere “\*” (asterisco) para retornar todas as colunas de uma tabela:

**SELECT \* FROM TABELA;**

O exemplo abaixo recupera as colunas NOME e EMAIL de todos os registros da tabela TAB\_COLABORADOR:

**SELECT NOME, EMAIL FROM TAB\_COLABORADOR;**

Resultado da execução do comando SQL:



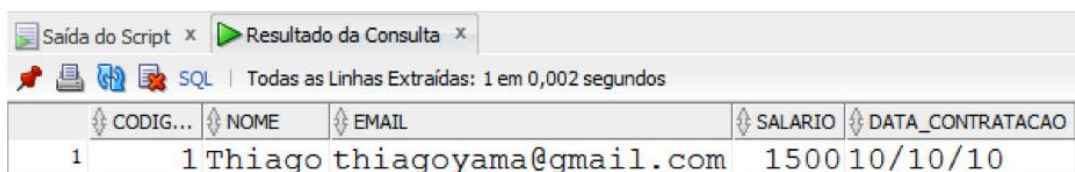
	NOME	EMAIL
1	Thiago	thiagoyama@gmail.com

Figura 7.11 – Resultado da execução do comando select  
Fonte: Fiap (2016)

Para recuperar todas as colunas da tabela podemos utilizar o comando:

**SELECT \* FROM TAB\_COLABORADOR;**

O resultado da execução será:



	CODIG...	NOME	EMAIL	SALARIO	DATA_CONTRATAAO
1	1	Thiago	thiagoyama@gmail.com	1500	10/10/10

Figura 7.12 – Resultado da execução  
Fonte: Fiap (2016)

Por enquanto, a nossa busca sempre retorna todos os registros. Imagine se uma tabela armazena milhares de registros ou se precisamos recuperar um registro específico da tabela. Por exemplo: um colaborador pelo seu código, ou todos os colaboradores que recebem um salário maior do que um determinado valor.

Para isso, precisamos adicionar filtros nas nossas buscas. Esses filtros permitem recuperar os registros de acordo com condições, além de trazer registros mais específicos, também utiliza menos capacidade processamento. Observe o exemplo abaixo:

- Buscar todos os colaboradores contratados em um determinado ano.
- Buscar todos os colaboradores que têm um salário menor do que um valor.
- Buscar todos os colaboradores que foram contratados em um ano específico e têm um salário menor do que um valor.

Dessa forma, precisamos colocar mais uma cláusula no comando SELECT: a cláusula WHERE:

**SELECT \* FROM TABELA WHERE CONDICA01;**

Para criar uma condição, precisamos utilizar os operadores relacionais:

Tabela 7.1 – TABELA WHERE CONDICA01

Operador	Significado	Exemplo
=	Igual	SALARIO = 1500
<	Menor que	SALARIO < 1500
>	Maior que	SALARIO > 1500
<=	Menor ou igual	SALARIO <= 1500
>=	Maior ou igual	SALARIO >= 1500
<> ou !=	Diferente	SALARIO <>1500 ou SALARIO !=1500
BETWEEN	Entre	SALARIO BETWEEN 1500 AND 5000

Fonte: Fiap (2016)

Exemplos:

- Buscar todos os colaboradores que têm salário maior do que R\$1.000,00:

**SELECT \* FROM TAB\_COLABORADOR WHERE SALARIO > 1000;**

- Buscar o colaborador que possui o código igual a 1:

```
SELECT * FROM TAB_COLABORADOR WHERE  
CODIGO_COLABORADOR = 1;
```

- Buscar todos os colaboradores que possuem o nome igual a “Thiago”:

```
SELECT * FROM TAB_COLABORADOR WHERE NOME = 'Thiago';
```

Às vezes são necessárias duas condições para filtrar uma busca. Por exemplo: buscar todos os colaboradores que têm o salário menor do que R\$ 1.500,00 e que foram contratados antes do ano 2000. Para isso, será necessário utilizar os operadores lógicos:

Operador	Descrição
AND	Retorna Verdadeiro se ambas as condições forem verdadeiras
OR	Retorna Verdadeiro se ao menos uma condição for verdadeira
NOT	Retorna Verdadeiro se a condição for falsa
OPERADORES LÓGICOS	

Quadro 7.2 – Operadores lógicos  
Fonte: Fiap (2016)

Exemplos:

- Buscar todos os colaboradores que têm um salário menor do que R\$3.000,00 e foram contratados antes do dia 10/10/2015:

```
SELECT * FROM TAB_COLABORADOR WHERE SALARIO < 3000  
AND DATA_CONTRATACAO < TO_DATE('10/10/2015','dd/mm/yyyy');
```

- Buscar todos os colaboradores que têm um salário menor do que R\$3.000,00 ou foram contratados antes do dia 10/10/2015:

```
SELECT * FROM TAB_COLABORADOR WHERE SALARIO < 3000 OR  
DATA_CONTRATACAO < TO_DATE('10/10/2015','dd/mm/yyyy');
```

### 7.2.3 Atualizando valores na tabela

Para atualizar os valores de um registro de uma tabela, precisamos utilizar o comando **UPDATE**. Esse comando permite a alteração do conteúdo de um ou mais campos (colunas) pertencentes a um ou mais registros (linhas) de uma tabela.

A sintaxe básica é:

**UPDATE TABELA SET COLUNA1 = NOVO\_VALOR1, COLUNA2 = NOVO\_VALOR2 WHERE CONDIÇÃO.**

A cláusula WHERE é utilizada para restringir os registros que serão alterados, pois caso não seja utilizada a cláusula WHERE, todos os registros serão alterados.

Exemplo:

**UPDATE TAB\_COLABORADOR SET NOME = 'Thiago Yamamoto' WHERE CODIGO\_COLABORADOR = 1;**

No exemplo acima, o colaborador com o código igual a 1 teve o seu nome alterado para 'Thiago Yamamoto'. Após a execução do comando UPDATE, foi realizado uma busca (SELECT) para visualizar a alteração na tabela.



CODIGO...	NOME	EMAIL	SALARIO	DATA_CONTRATACAO
1	Thiago Yamamoto	thiagoyama@gmail.com	1500	14/08/15

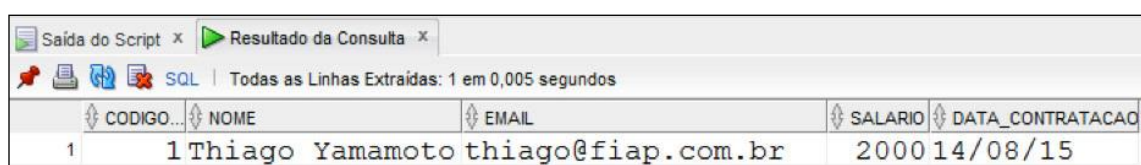
Figura 7.13 – Resultado da alteração do registro na tabela

Fonte: Fiap (2016)

É possível alterar várias colunas em um mesmo comando UPDATE, basta separar as colunas por vírgula:

**UPDATE TAB\_COLABORADOR SET EMAIL = 'thiago@fiap.com.br', SALARIO = 2000 WHERE CODIGO\_COLABORADOR = 1;**

O comando acima altera as colunas EMAIL e SALARIO do colaborador com código igual a 1. O resultado da alteração pode ser visualizado na figura abaixo:



CODIGO...	NOME	EMAIL	SALARIO	DATA_CONTRATACAO
1	Thiago Yamamoto	thiago@fiap.com.br	2000	14/08/15

Figura 7.14 – Atualização de um registro da tabela TAB\_COLABORADOR

Fonte: Fiap (2016)

### 7.2.4 Remoção de registros de uma tabela (Delete)

Para remover um registro de uma tabela é utilizado o comando **delete**:

**DELETE FROM TABELA WHERE CONDIÇÃO;**

Esse comando também precisa de condição, pois sem ela, todos os registros da tabela serão apagados.

Exemplo:

**DELETE FROM TAB\_COLABORADOR WHERE CODIGO\_COLABORADOR = 1;**

No exemplo acima foi removido o registro que possui o CODIGO\_COLABORADOR igual a 1. Agora que conhecemos um pouco dos comandos básicos do SQL, vamos começar a programação Java!

### 7.2.5 Conectando a base de dados

Primeiro, vamos criar um novo projeto para manipular a base de dados. Para isso, vá à opção “File” do menu e escolha “New” e depois “Java Project”, conforme a figura abaixo:

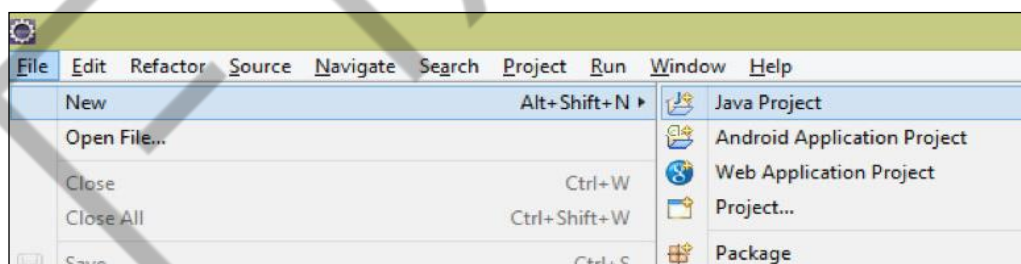


Figura 7.15 – Criando um novo projeto Java  
Fonte: Fiap (2016)

Dê um nome ao projeto e finalize o processo.

O próximo passo é configurar o ambiente. Como explicado anteriormente, precisamos do driver JDBC do banco de dados Oracle. Dessa forma, vamos adicionar o arquivo .jar no nosso projeto.

Para isso, crie uma pasta chamada lib e adicione o driver do Oracle, conforme os seguintes passos:

Crie uma nova pasta dentro do projeto.

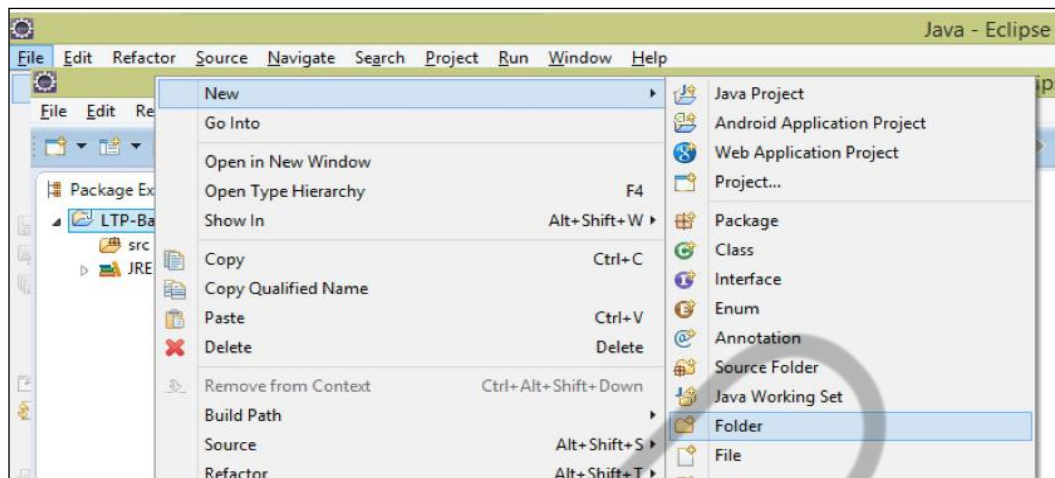


Figura 7.16 – Criando uma pasta para o driver do Oracle – Parte 1  
Fonte: Fiap (2016)

Dê um nome para a pasta (como sugestão “lib”) e finalize o processo.

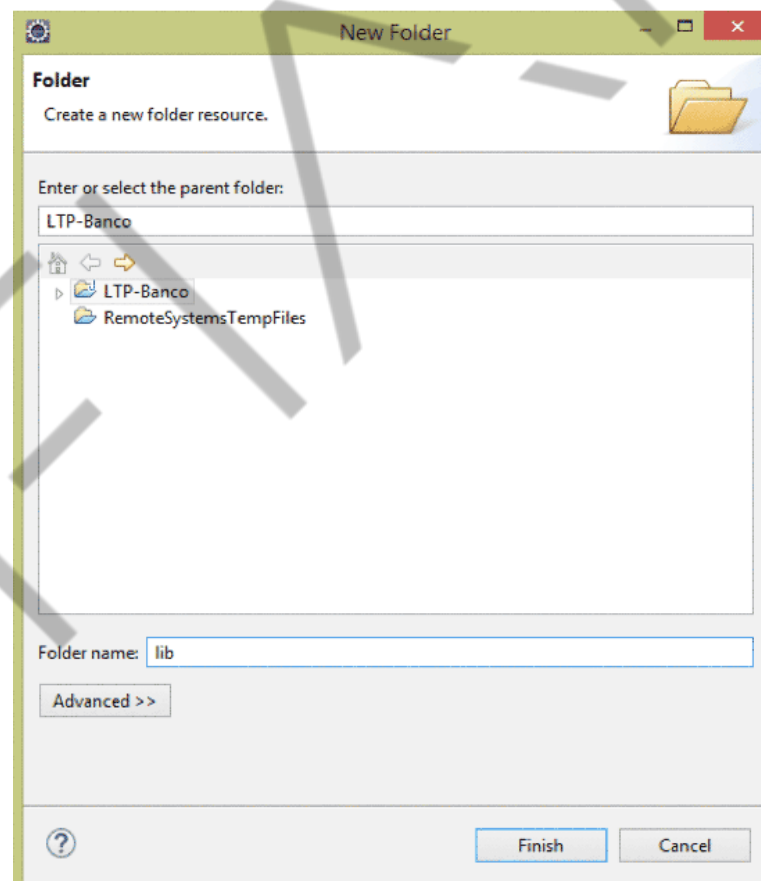


Figura 7.17 – Criando uma pasta para o driver do Oracle – Parte 2  
Fonte: Fiap (2016)

Agora copie o driver do Oracle para dentro da pasta e adicione no build path do projeto, para que as classes e interfaces do driver fiquem disponíveis para o uso dentro do projeto. Para adicionar no build path, clique com o botão direito do mouse no driver e escolha: “Build Path” -> “Add to Build Path”, conforme a figura abaixo:

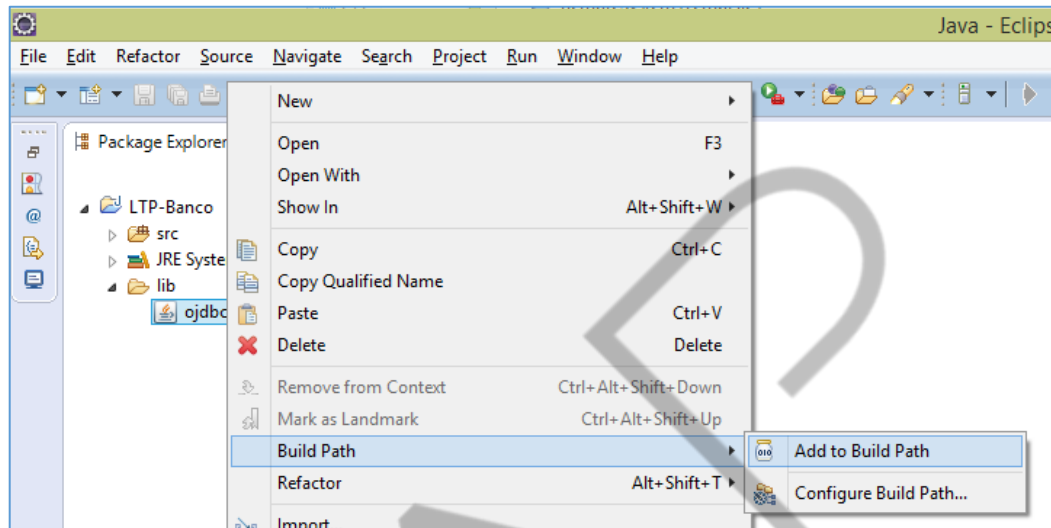


Figura 7.18 – Adicionando o driver do Oracle no projeto  
Fonte: Fiap (2016)

Agora sim, estamos prontos para começar a desenvolver!

Primeiro vamos estabelecer uma conexão com a base de dados. Precisamos criar um objeto do tipo **Connection** que representará a conexão. Depois, estaremos aptos a realizar qualquer operação (Cadastrar, Atualizar, Apagar e Buscar) na base de dados.

Abaixo, apresentamos uma classe de teste que realiza a conexão com o banco de dados:

```
package br.com.fiap.teste;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class TesteView {

    public static void main(String[] args) {

        try {
```

```
//Registra o Driver

Class.forName("oracle.jdbc.driver.OracleDriver");

//Abre uma conexão
Connection conexao =
DriverManager.getConnection(

    "jdbc:oracle:thin:@192.168.60.15:1521:ORCL", "OPS$PF0392",
    "123456");

System.out.println("Conectado!");

//Fecha a conexão
conexao.close();

//Tratamento de erro
} catch (SQLException e) {
    System.err.println("Não foi possível
conectar no Banco de Dados");
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    System.err.println("O Driver JDBC não foi
encontrado!");
    e.printStackTrace();
}
}
}
```



O primeiro passo para obtermos uma conexão é registrar o driver que será utilizado. Depois, através do método **DriverManager.getConnection()** obtemos uma conexão com o banco de dados. Os parâmetros desse método são:

- A URL para acessar o banco de dados.
- Usuário do banco de dados.
- Senha do banco de dados.

Após obter a conexão é possível enviar qualquer comando SQL para realizar alguma operação no banco de dados. Veremos isso daqui a pouco. Depois de enviar as instruções, devemos fechar a conexão utilizando o método **close()**.

Observe que foi realizado o tratamento de dois possíveis erros. O catch do **SQLException** trata o erro de conexão com o banco de dados. Já a exceção **ClassNotFoundException** pode ocorrer quando a aplicação não encontra o driver do oracle.

Agora vamos discutir um pouco sobre cada uma das classes e interfaces que iremos utilizar para se conectar e realizar as operações na base de dados:

#### **DriverManager:**

Determina qual driver de conexão será utilizado. Funciona como uma ponte entre o JDBC e o driver escolhido.

Quando escrevemos o código **Class.forName("localização do driver");** estamos definindo a classe (e seu pacote) principal da biblioteca do driver. Esta informação pode ser obtida na documentação do fabricante do driver.

Nas versões atuais do Java não é necessário realizar esse processo, pois a partir da String de conexão a JVM reconhece qual driver deve ser utilizado.

#### **Connection:**

Interface do JDBC que representa a conexão com a base de dados. Através do método **getConnection()** da classe **DriverManager** podemos obter um objeto do tipo da interface **Connection** que representa a conexão. Esse método recebe três parâmetros: a URL de conexão, usuário e senha para acesso ao banco.

A URL de conexão também é definida pelo fabricante do SGDB, seguindo o padrão JDBC. Abaixo, é apresentada um quadro com as principais classes do driver e URL de conexão para os principais SGDBs:

RDBMS	JDBC driver	URL string
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname: port Number/databaseName
JDBC DRIVERS E URL STRING		

Quadro 7.3 – JDBC Drivers e URL String.  
Fonte: Fiap (2016)

### 7.2.6 Statements

Para executar comandos SQLs no SGBDR, existem três objetos do tipo **Statement**:

- **Statement**: Utilizado para executar um comando SQL estático.
- **PreparedStatement**: Utilizado para executar um comando SQL que recebe um ou mais parâmetros.
- **Callable Statement**: Utilizado para chamar *stored procedures* ou *functions*.

Os principais métodos destas implementações são:

- **executeUpdate**: executa um comando SQL (INSERT, UPDATE, DELETE) e retorna o número de linhas afetadas.
- **executeQuery**: executa um comando SQL (SELECT) e retorna o(s) resultado(s) através de um objeto do tipo **ResultSet**.

Para recuperar o objeto do tipo **Statement**, utilizamos o método **createStatement()** da interface **Connection**:

Exemplos:

### Inclusão:

```
Statement stmt = conexao.createStatement();
stmt.executeUpdate("INSERT INTO
TAB_COLABORADOR(CODIGO_COLABORADOR, NOME, EMAIL, SALARIO,
DATA_CONTRATACAO) VALUES (SQ_COLABORADOR.NEXTVAL, 'Leandro',
'leandro@gmail.com', 1500,
TO_DATE('10/12/2009','dd/mm/yyyy'))");
```

### Alteração:

```
Statement stmt = conexao.createStatement();
stmt.executeUpdate("UPDATE TAB_COLABORADOR SET SALARIO =
5000 WHERE CODIGO_COLABORADOR = 1");
```

### Exclusão:

```
Statement stmt = conexao.createStatement();
stmt.executeUpdate("DELETE FROM TAB_COLABORADOR WHERE
CODIGO_COLABORADOR = 1");
```

### Busca:

```
Statement stmt = conexao.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM
TAB_COLABORADOR");
```

Observe que os comandos SQL são estáticos, ou seja, os valores já estão definidos diretamente na String. Quando precisamos de comandos SQL configuráveis, devemos utilizar a interface `PreparedStatement`.

Dessa forma, podemos parametrizar o comando SQL com o ponto de interrogação `?`, através dos métodos `setXXX()` é possível atribuir valores a esses parâmetros, evitando assim ataques do tipo SQL Injection.

É indicado utilizar o `PreparedStatement` para manipular a base de dados, sempre que possível, pois proporciona melhor performance e clareza do código fonte. Para obter o objeto `PreparedStatement` utilizamos o método **`prepareStatement()`** da interface **`Connection`**.

**Exemplo:**

```
PreparedStatement stmt = conexao.prepareStatement("INSERT
INTO TAB_COLABORADOR(CODIGO_COLABORADOR, NOME, EMAIL, SALARIO,
DATA_CONTRATACAO) VALUES (SQ_COLABORADOR.NEXTVAL, ?, ?, ?,
?)");
stmt.setString(1, "Thiago"); //Primeiro parâmetro (Nome)
stmt.setString(2, "thiago@gmail.com");//Segundo parâmetro
(Email)
stmt.setDouble(3, 5000); //Terceiro parâmetro (Salário)
//Instancia um objeto do tipo java.sql.Date com a data
atual
java.sql.Date data = new java.sql.Date(new
java.util.Date().getTime());
stmt.setDate(4,data);//Quarto parâmetro (data contratação)

stmt.executeUpdate();
```

Observe que no lugar dos valores dos parâmetros foi adicionado o ponto de interrogação ?, para posteriormente adicionarmos valores a eles através dos métodos setXXX, dependendo do tipo de dado. Esse método recebe dois parâmetros: o primeiro é a posição do sinal ?, que inicia-se em 1, já o segundo é o valor que será atribuído a essa posição.

Para cada uma das operações do SQL é possível utilizar o **PreparedStatement**:

**Alteração:**

```
PreparedStatement stmt = conexao.prepareStatement("UPDATE
TAB_COLABORADOR SET SALARIO = ? WHERE CODIGO_COLABORADOR =
?");
stmt.setDouble(1, 5000);
stmt.setInt(2, 100);
stmt.executeUpdate();
```

**Exclusão:**

```
PreparedStatement stmt = conexao.prepareStatement("DELETE
FROM TAB_COLABORADOR WHERE CODIGO_COLABORADOR = ?");
stmt.setInt(1, 1);
stmt.executeUpdate();
```

**Busca:**

```
PreparedStatement stmt = conexao.prepareStatement("SELECT  
* FROM TAB_COLABORADOR WHERE NOME = ?");  
stmt.setString(1, "Thiago");  
ResultSet result = stmt.executeQuery();
```

Agora que aprendemos como executar os principais comandos SQL através da linguagem Java, vamos abordar como recuperar as informações através da interface **ResultSet**.

**ResultSet:**

A interface **ResultSet** é responsável pelo conjunto de registros retornados de um comando SELECT do SQL. Através desse tipo de objeto podemos navegar por seus registros de forma sequencial, dessa forma, precisamos chamar o método **next** para mover o cursor para o próximo registro. Esse método retorna **false** quando não conseguir ir para o próximo registro, que caracteriza o final dos dados. Inicialmente o cursor está posicionado antes do primeiro registro.

As colunas do registro podem ser acessadas através de um índice que representa a posição da coluna (inicia em 1) ou através do próprio nome da coluna.

**Principais métodos:**

- **next:** move o cursor para a próxima linha.
- **getInt:** retorna os dados da coluna designada como um **int** do Java.
- **getString:** retorna os dados da coluna designada como uma **String** do Java.
- **getBoolean:** retorna os dados da coluna designada como um **boolean** do Java.
- **getDouble:** retorna os dados da coluna designada como um **double** do Java.

Exemplo:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class TesteView {

    public static void main(String[] args) {
        try {
            //Registra o Driver
            Class.forName("oracle.jdbc.driver.OracleDriver");

            //Abre uma conexão
            Connection conexao = DriverManager.getConnection(
                "jdbc:oracle:thin:@oracle.fiap.com.br:1521:ORCL",
                "OPS$PF0392", "123456");

            System.out.println("Conectado!");

            PreparedStatement stmt = conexao.prepareStatement("SELECT
* FROM TAB_COLABORADOR WHERE NOME = ?");
            stmt.setString(1, "Thiago");
            ResultSet result = stmt.executeQuery();

            //Percorre todos os registros encontrados
            while (result.next()) {
                //Recupera os valores de cada coluna
                int codigo =
result.getInt("CODIGO_COLABORADOR");
                String nome = result.getString("NOME");
                String email = result.getString("EMAIL");
                double salario = result.getDouble("SALARIO");
```

```
        java.sql.Date data =
result.getDate("DATA_CONTRATACAO");

        //Exibe as informações do registro
        System.out.println(codigo + " " + nome + " " +
email + " " + salario + " " + data);
    }

    //Fecha a conexão
    conexao.close();

    //Tratamento de erro
    } catch (SQLException e) {
        System.err.println("Não foi possível conectar no
Banco de Dados");
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        System.err.println("O Driver JDBC não foi
encontrado!");
        e.printStackTrace();
    }
}
}
```

## 7.2.7 CallableStatement

Utilizado para chamar *stored procedures* ou *functions* de forma padronizada para todas as bases de dados. É possível chamar uma *stored procedure* utilizando ou não parâmetro de resultado.

Sintaxe básica para chamada de *stored procedure*:

- Com parâmetro de resultado:

CallableStatement cs = conexao.prepareCall("{call proc(?)}");

- Sem parâmetro de resultado:

CallableStatement cs = conexao.prepareCall("{call proc()}");

Para definir os parâmetros de entrada utilizamos o mesmo padrão do **PreparedStatement**, já para os parâmetros de saída são definidos através do método **registerOutParameter**.

Exemplo de chamada de uma *Stored Procedure* com parâmetros de entrada e saída:

```
//Cria o CallableStatement
CallableStatement cs = conexao.prepareStatement("{call
SP_Contar_Colaboradores(?,?)}");

//Define o tipo do parâmetro de saída (primeiro ?)
cs.registerOutParameter(1, java.sql.Types.INTEGER);

//Define o valor do parâmetro de entrada (segundo ?)
cs.setDouble(2, 1500);

//Executa a procedure
cs.executeUpdate();

//Recupera o valor do parâmetro de saída
int total = cs.getInt(1);
System.out.println("Total de colaboradores com salário
maior que 1500: " + total);
```

Exemplo de uma *Stored Procedure* que retorna o resultado de um SELECT:

```
//Cria o CallableStatement
CallableStatement cs = conexao.prepareStatement("{call
SP_Retornar_Todos_Colaboradores(?,?)}");

//Define o valor do parâmetro de entrada
cs.setDouble(1, 1500);
```



```
//Define o tipo do parâmetro de saída
cs.registerOutParameter(2, OracleTypes.CURSOR);

//Executa a procedure
cs.execute();

//Recupera o valor do parâmetro de saída
ResultSet cursor = (ResultSet) cs.getObject(2);

//Percorre todos os registros encontrados
while (cursor.next()) {
    //Recupera os valores de cada coluna
    int codigo = cursor.getInt("CODIGO_COLABORADOR");
    String nome = cursor.getString("NOME");
    System.out.println(codigo + " " + nome);
}
```

Agora que estudamos as principais Classes e Interfaces do JDBC, vamos implementar o nosso exemplo!

Aproveitando as tabelas e o projeto criados anteriormente, vamos começar desenvolvendo o Java Bean **Colaborador** que irá representar a nossa tabela do banco de dados.

Essa classe deve possuir atributos que representam cada coluna da tabela. Utilizamos o encapsulamento, deixando os atributos com o modificador de acesso **private** e disponibilizamos os métodos assessores (gets e sets). Foi implementado também o construtor padrão e com parâmetros. Para armazenar a data de contratação utilizamos a classe java.util.**Calendar**.

```
package br.com.fia.bean;
import java.util.Calendar;
public class Colaborador {

    private int codigo;

    private String nome;

    private String email;

    private double salario;

    private Calendar dataContratacao;

    public Colaborador(int codigo, String nome, String
email, double salario,
        Calendar dataContratacao) {
        super();
        this.codigo = codigo;
        this.nome = nome;
        this.email = email;
        this.salario = salario;
        this.dataContratacao = dataContratacao;
    }

    public Colaborador() {
        super();
    }

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }
}
```

```
}

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public double getSalario() {
        return salario;
    }

    public void setSalario(double salario) {
        this.salario = salario;
    }

    public Calendar getDataContratacao() {
        return dataContratacao;
    }

    public void setDataContratacao(Calendar
dataContratacao) {
        this.dataContratacao = dataContratacao;
    }
}
```

Essa classe possui grande importância por facilitar o processo de passagem de valores do banco de dados para a aplicação Java e vice-versa.

Agora vamos criar uma classe que será responsável por fornecer uma conexão com o banco de dados. Observe que para cada operação no banco de dados (INSERT, UPDATE, SELECT e DELETE) foi necessário primeiro obter uma conexão. Com essa classe, não vamos precisar escrever código repetido, utilizando assim os conceitos de orientação a objetos.

```
package br.com.fiap.jdbc;

import java.sql.Connection;
import java.sql.DriverManager;

public class EmpresaDBManager {

    public static Connection obterConexao() {
        Connection conexao = null;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");

            conexao = DriverManager.getConnection(
                "jdbc:oracle:thin:@oracle.fiap.com.br:1521:ORCL",
                "OPS$XXXX", "XXXX");
        } catch (Exception e) {
            e.printStackTrace();
        }
        return conexao;
    }
}
```

Observe que essa classe possui somente um método estático que cria e retorna uma conexão com o banco. O método é estático para não precisar de uma instância para ser invocada, bastando referenciá-la através do nome da classe: **EmpresaDBManager.obterConexao();**

Agora estamos prontos para desenvolver a classe Java responsável pela manipulação da tabela TAB\_COLABORADOR. A classe possui a nomenclatura **ColaboradorDAO**, pois segue um padrão de projeto chamado de DAO (Data Access Object), que é responsável pelo código de acesso aos dados, centralizando assim a implementação dessa responsabilidade no projeto.

Vamos falar sobre Padrões de Projetos, suas implementações e benefícios um pouco mais para frente no curso.

Crie a classe **ColaboradorDAO**, conforme a listagem abaixo:

```
package br.com.fiap.dao;

public class ColaboradorDAO {

    private Connection conexao;

}
```

A classe DAO deve possuir um atributo para armazenar o objeto que representa a conexão com o banco de dados.

Agora vamos desenvolver o primeiro método que será responsável por cadastrar um colaborador na base de dados. Para isso, ele deve receber um objeto do tipo Colaborador para ser inserido no banco:

```
public class ColaboradorDAO {

    private Connection conexao;

    public void cadastrar(Colaborador colaborador) {
        PreparedStatement stmt = null;
    }

}
```

```
        try {
            conexao = EmpresaDBManager.obterConexao();
            String sql = "INSERT INTO
TAB_COLABORADOR(CODIGO_COLABORADOR, NOME, EMAIL, SALARIO,
DATA_CONTRATACAO) VALUES (SQ_COLABORADOR.NEXTVAL, ?, ?, ?,
?)";

            stmt = conexao.prepareStatement(sql);
            stmt.setString(1, colaborador.getNome());
            stmt.setString(2, colaborador.getEmail());
            stmt.setDouble(3,
colaborador.getSalario());
            java.sql.Date data = new
java.sql.Date(colaborador.getDataContratacao().getTimeInMillis
());

            stmt.setDate(4, data);

            stmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                stmt.close();
                conexao.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Primeiro, obtemos a conexão com o banco de dados através do método `obterConexao()` da classe `EmpresaDBManager`. Depois, definimos em uma `String` o comando SQL. Criamos o `PreparedStatement` e passamos os valores para cada um de seus parâmetros. Por fim, executamos o comando no Banco de dados. Utilizamos o bloco `Try-catch` para tratar possíveis exceções e no bloco `finally` (que sempre é executado) fechamos o `PreparedStatement` e a conexão.

Não se esqueça dos *imports*! Um atalho do eclipse é o `CTR + SHIF + o`, para fazer o import automático.

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import br.com.fiap.bean.Colaborador;
import br.com.fiap.jdbc.EmpresaDBManager;
```

Após implementar uma funcionalidade no DAO, é indicado desenvolver uma classe de teste, para verificar se está tudo correto.

Para isso, crie uma classe com o método `main`, conforme a listagem abaixo:

```
package br.com.fiap.teste;

import java.util.Calendar;

import br.com.fiap.bean.Colaborador;
import br.com.fiap.dao.ColaboradorDAO;

public class TesteCadastro {

    public static void main(String[] args) {
        //Instancia o DAO
        ColaboradorDAO dao = new ColaboradorDAO();

        //Instancia o Colaborador
        Colaborador colaborador = new Colaborador();
    }
}
```

```
colaborador.setNome("Pedro");
colaborador.setEmail("pedro@fiap.com.br");
colaborador.setSalario(5000);

colaborador.setDataContratacao(Calendar.getInstance());

//Cadastra no banco de dados
dao.cadastrar(colaborador);

System.out.println("Cadastrado!");
}

}
```

Essa classe cria uma instância da classe ColaboradorDAO e do Colaborador. E através do método cadastrar do ColaboradorDAO é realizado o cadastro no banco de dados.

Execute e verifique no banco de dados se o colaborador foi cadastrado com sucesso!

Agora vamos implementar a próxima funcionalidade no DAO: **Listar**. Para isso, volte à classe ColaboradorDAO e crie o novo método:

```
public List<Colaborador> listar() {
    //Cria uma lista de colaboradores
    List<Colaborador> lista = new
    ArrayList<Colaborador>();
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conexao = EmpresaDBManager.obterConexao();
        stmt = conexao.prepareStatement("SELECT *
FROM TAB_COLABORADOR");
        rs = stmt.executeQuery();
    }
```



```
        //Percorre todos os registros encontrados
        while (rs.next()) {
            int codigo =
rs.getInt("CODIGO_COLABORADOR");

            String nome = rs.getString("NOME");
            String email = rs.getString("EMAIL");
            double salario =
rs.getDouble("SALARIO");

            java.sql.Date data =
rs.getDate("DATA_CONTRATACAO");

            Calendar dataContratacao =
Calendar.getInstance();

            dataContratacao.setTimeInMillis(data.getTime());
            //Cria um objeto Colaborador com as
informações encontradas
            Colaborador colaborador = new
Colaborador(codigo, nome, email, salario, dataContratacao);
            //Adiciona o colaborador na lista
            lista.add(colaborador);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            stmt.close();
            rs.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

return lista;
}
```

Esse método retorna uma lista com todos os colaboradores cadastrados. Não se esqueça de adicionar os imports do List, ArrayList e ResultSet:

```
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;
```

Para cada registro no ResultSet é instanciado um objeto Colaborador para armazenar as informações encontradas. Esse objeto é adicionado à lista que será retornada.

Agora vamos para a classe de teste! Para ficar mais organizado, criaremos uma nova classe de teste que será responsável pelo teste da listagem:

```
package br.com.fiap.teste;

import java.util.List;
import br.com.fiap.bean.Colaborador;
import br.com.fiap.dao.ColaboradorDAO;

public class TesteListagem {

    public static void main(String[] args) {

        ColaboradorDAO dao = new ColaboradorDAO();

        List<Colaborador> lista = dao.listar();
        for (Colaborador item : lista) {
            System.out.println(item.getCodigo() + " " +
item.getNome() + " " + item.getEmail() + " " +
item.getSalario() + " " +
item.getDataContratacao().getTime());
        }
    }
}
```

```
}
```

Essa classe instanciará o ColaboradorDAO e chamar o método listar() para receber a lista de colaboradores cadastrados no banco de dados. Depois, implementamos um laço de repetição foreach para percorrer toda a lista e imprimir os valores dos atributos do colaborador.

A próxima funcionalidade será remover um colaborador da base de dados:

```
public void remover(int codigo) {  
    PreparedStatement stmt = null;  
  
    try {  
        conexao = EmpresaDBManager.obterConexao();  
        String sql = "DELETE FROM TAB_COLABORADOR  
WHERE CODIGO_COLABORADOR = ?";  
        stmt = conexao.prepareStatement(sql);  
        stmt.setInt(1, codigo);  
        stmt.executeUpdate();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    } finally {  
        try {  
            stmt.close();  
            conexao.close();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Esse método, que deve ser implementado na classe ColaboradorDAO, recebe o código do colaborador que será excluído do banco de dados.

A nova classe de teste pode ser escrita dessa forma:

```
package br.com.fiap.teste;

import br.com.fiap.dao.ColaboradorDAO;

public class TesteRemocao {

    public static void main(String[] args) {
        ColaboradorDAO dao = new ColaboradorDAO();
        //Remove o colaborador com código 1
        dao.remover(1);
    }
}
```

A classe de teste instancia o ColaboradorDAO para chamar o método remover, passando o ID do colaborador que será excluído do banco de dados.

Para finalizar o CRUD precisamos implementar o método de atualização. Porém, antes vamos desenvolver o método buscar por código, para recuperar o colaborador que será atualizado.

Na classe ColaboradorDAO adicione o método buscarPorId que recebe o código do colaborador:

```
Colaborador colaborador = null;
PreparedStatement stmt = null;
ResultSet rs = null;
try {
    conexao = EmpresaDBManager.obterConexao();
    stmt = conexao.prepareStatement("SELECT *
FROM TAB_COLABORADOR WHERE CODIGO_COLABORADOR = ?");
    stmt.setInt(1, codigoBusca);
    rs = stmt.executeQuery();
    if (rs.next()) {
```

```
        int codigo =  
rs.getInt("CODIGO_COLABORADOR");  
        String nome = rs.getString("NOME");  
        String email = rs.getString("EMAIL");  
        double salario =  
rs.getDouble("SALARIO");  
        java.sql.Date data =  
rs.getDate("DATA_CONTRATACAO");  
        Calendar dataContratacao =  
Calendar.getInstance();  
        dataContratacao.setTimeInMillis(data.getTime());  
        colaborador = new Colaborador(codigo,  
nome, email, salario, dataContratacao);  
    }  
    } catch (SQLException e) {  
        e.printStackTrace();  
    } finally {  
        try {  
            stmt.close();  
            rs.close();  
            conexao.close();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
    }  
    return colaborador;  
}
```

Observe que após a execução da Query é acionado o método `next()` do `ResultSet` para verificar se existe algum registro, em caso positivo, são recuperados todos os valores do registro e instanciado um `Colaborador` para armazenar essas informações.

Agora, o último método, o de atualização:

```
public void atualizar(Colaborador colaborador) {  
    PreparedStatement stmt = null;  
  
    try {  
        conexao = EmpresaDBManager.obterConexao();  
        String sql = "UPDATE TAB_COLABORADOR SET  
NOME = ?, EMAIL = ?, SALARIO = ?, DATA_CONTRATACAO = ? WHERE  
CODIGO_COLABORADOR = ?";  
        stmt = conexao.prepareStatement(sql);  
        stmt.setString(1, colaborador.getNome());  
        stmt.setString(2, colaborador.getEmail());  
        stmt.setDouble(3,  
colaborador.getSalario());  
        java.sql.Date data = new  
java.sql.Date(colaborador.getDataContratacao().getTimeInMillis  
());  
        stmt.setDate(4, data);  
        stmt.setInt(5, colaborador.getCodigo());  
  
        stmt.executeUpdate();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    } finally {  
        try {  
            stmt.close();  
            conexao.close();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Foram adicionados na query todos os atributos do Colaborador para ser atualizado, menos o código, pois ele nunca deve ser alterado. Não se esqueça de adicionar a cláusula WHERE para modificar somente o colaborador correto.

A classe de teste deve primeiro recuperar um colaborador, utilizando o método buscarPorId, depois devemos alterar os valores dos atributos do colaborador e chamar o método atualizar:

```
package br.com.fiap.teste;

import br.com.fiap.bean.Colaborador;
import br.com.fiap.dao.ColaboradorDAO;

public class TesteAlteracao {

    public static void main(String[] args) {

        ColaboradorDAO dao = new ColaboradorDAO();
        //Recupera o colaborador com código 1
        Colaborador colaborador = dao.buscarPorId(1);
        //Imprime os valores do colaborador
        System.out.println(colaborador.getCodigo() + " "
            + colaborador.getNome() + " " +
colaborador.getEmail() + " "
            + colaborador.getSalario() + " "
            +
colaborador.getDataContratacao().getTime());
        //Altera os valores de alguns atributos do
colaborador
        colaborador.setSalario(1500);
        colaborador.setEmail("teste@fiap.com.br");
        //Atualiza no banco de dados
        dao.atualizar(colaborador);
    }
}
```

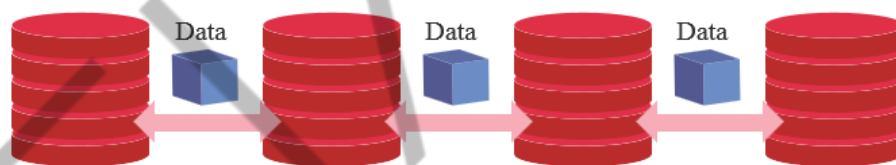
### 7.3 Controle transacional

Toda comunicação com banco de dados requer cuidados com acessos paralelos e integridade dos dados. Para isso, todo SGBD e linguagem de programação possuem tratamento específico para permitir a segurança e integridade dos dados, assim como no JDBC.

#### 7.3.1 Transação

Uma transação é uma unidade que preserva a consistência de informações no banco de dados.

Podemos dizer também que transações são unidades atômicas de operações: “Em ciência da computação, uma transação atômica é uma operação, ou conjunto de operações, em uma base de dados, ou em qualquer outro sistema computacional, que deve ser executada completamente em caso de sucesso, ou ser abortada completamente em caso de erro”.



Transacionalidade  
Figura 7.19 – Processo transacional  
Fonte: Fiap (2016)

Um exemplo clássico da necessidade de uma “transação atômica” é a transferência entre duas contas bancárias. No momento de uma transferência de valores de uma conta **A** para uma conta **B**, que envolve pelo menos uma operação de ajuste no saldo para cada conta, dessa forma, as duas contas devem sofrer alteração no saldo ou nenhuma das duas contas. Caso o computador responsável pela operação é desligado no meio da operação, é esperado que o saldo de ambas as contas não tenha se alterado. Neste caso, são utilizados sistemas que suportam transações atômicas.



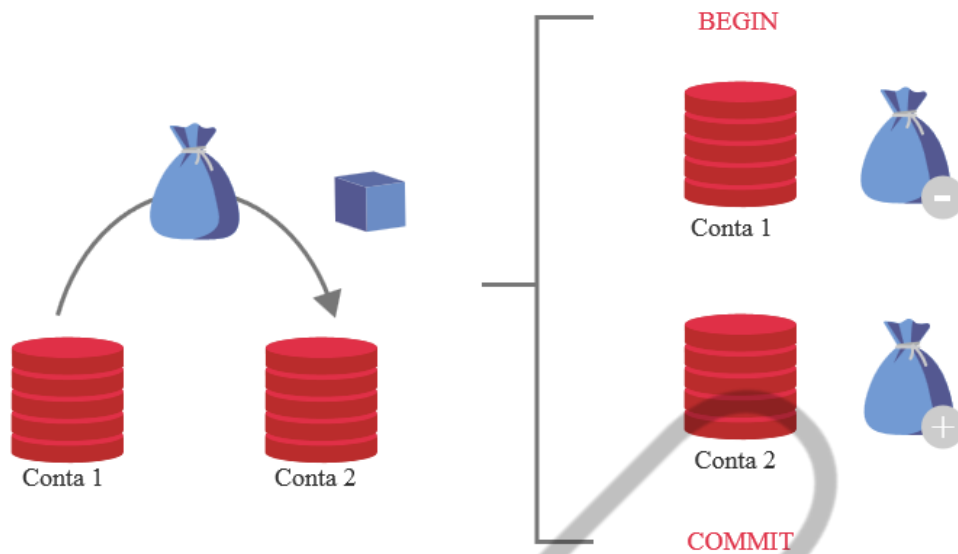


Figura 7.20 – Exemplo de transação atômica.  
Fonte: Fiap (2016)

Uma transação pode ser representada pela sigla ACID:

- **Atomicidade(Atomicity):** atômico, tudo (commit) ou nada (rollback).
- **Consistência(Consistency):** toda transação executada deve seguir as regras de integridade do banco de dados, mantendo assim a consistência da base de dados.
- **Isolamento(Isolation):** garante que nenhuma transação seja interferida por outra até que esta seja completada.
- **Durabilidade(Durability):** garante que as informações gravadas no banco de dados durem de forma imutável até que outra transação de atualização ou remoção a afete.

As operações possíveis para o controle transacional são: **COMMIT** e **ROLLBACK**:

- **COMMIT:** todas as operações envolvidas em uma unidade de processo executam com sucesso, efetivando assim as alterações na base de dados.
- **ROLLBACK:** todas as operações envolvidas em uma unidade de processo não executam corretamente, não efetuando assim as operações das transações na base de dados.

Por padrão, toda conexão com JDBC está configurada para realizar o **COMMIT** automaticamente, ou seja, o COMMIT é realizado de forma automática após a execução de um comando SQL.

Para desabilitar essa configuração, utilizamos o método **setAutoCommit(false)**, da interface **Connection**.

Exemplo:

```
try {  
    //Desabilita o autocommit  
    conexao.setAutoCommit(false);  
  
    //Primeira transação - Atualiza o salário  
    PreparedStatement stmt =  
conexao.prepareStatement("UPDATE TAB_COLABORADOR SET SALARIO =  
? WHERE CODIGO_COLABORADOR = ?");  
    stmt.setDouble(1, 5000);  
    stmt.setInt(2, 1);  
    stmt.executeUpdate();  
  
    //Segunda transação - Atualiza o e-mail  
    PreparedStatement stmt2 =  
conexao.prepareStatement("UPDATE TAB_COLABORADOR SET EMAIL = ?  
WHERE CODIGO_COLABORADOR = ?");  
    stmt2.setString(1, "teste@teste.com.br");  
    stmt2.setInt(2, 1);  
    stmt2.executeUpdate();  
  
    //Efetiva as duas transações  
    conexao.commit();  
  
} catch (SQLException se) {  
    //Não efetiva as duas transações  
    conexao.rollback();  
}
```

No exemplo acima é possível observar que desabilitamos o COMMIT automático através da instrução **conexao.setAutoCommit(false)**, dessa forma, todo comando que modifica a base de dados (atualização, inclusão e exclusão) só será efetivado caso o comando COMMIT seja chamado. Assim sendo, as duas instruções de atualização só serão efetivadas caso não ocorram erros, após o comando **conexao.commit()**.

Caso ocorra algum erro, o comando **conexao.rollback()** é executado para desfazer as transações e assim não modificar as informações do banco de dados.

#### 7.4 Design patterns

Um **Padrão de Projeto de Software**, também conhecido pelo termo em inglês **Design Pattern**, descreve uma solução geral reutilizável para um problema recorrente no desenvolvimento de sistemas. Não é um código final, é uma descrição ou modelo de como resolver o problema que ocorre em vários sistemas de diferentes áreas de atuação ou plataforma de desenvolvimento.

Foram desenvolvidos vários padrões de projetos, o livro que tornou os padrões mais populares e conhecidos foi escrito por quatro autores e se chama **Design Patterns: Elements of Resuable Object-Oriented Software**. Os autores desses livros são conhecidos como “Gangue dos Quatro” (Gang of Four) ou simplesmente “GOF”.

Utilizar padrões de projetos na implementação de um programa implica em algumas vantagens como:

- **Facilidade de comunicação:** os padrões possuem nomes, os quais descrevem uma solução que deve ser de conhecimento entre os desenvolvedores.
- **Credibilidade:** padrões de projetos são utilizados constantemente nas implementações de sistemas, dessa forma, são amplamente testadas e aprovadas.
- **Facilidade de manutenção:** além do desenvolvimento, a manutenção é outra etapa da vida de um sistema que devemos nos preocupar. Padrões

de projetos tendem a reduzir o acoplamento entre os componentes, o que facilita a manutenção de um sistema.

A seguir, veremos os principais Design Patterns para implementação do *backend* de um sistema.

#### 7.4.1 Data Access Object (DAO)

Esse padrão de projeto abstrai e encapsula todo o acesso à base de dados. A ideia é criar uma classe Java que separa totalmente a lógica de acesso e manipulação de dados da aplicação. Essa classe denominada DAO é a camada que separa a aplicação do banco de dados, ela possui todo o código de acesso ao banco de dados, tornando mais fácil os testes e manutenção.

Dessa forma, devemos desenvolver uma classe DAO para cada entidade, a fim de separar a responsabilidade do acesso à base de dados da entidade a seus respectivos DAOs:

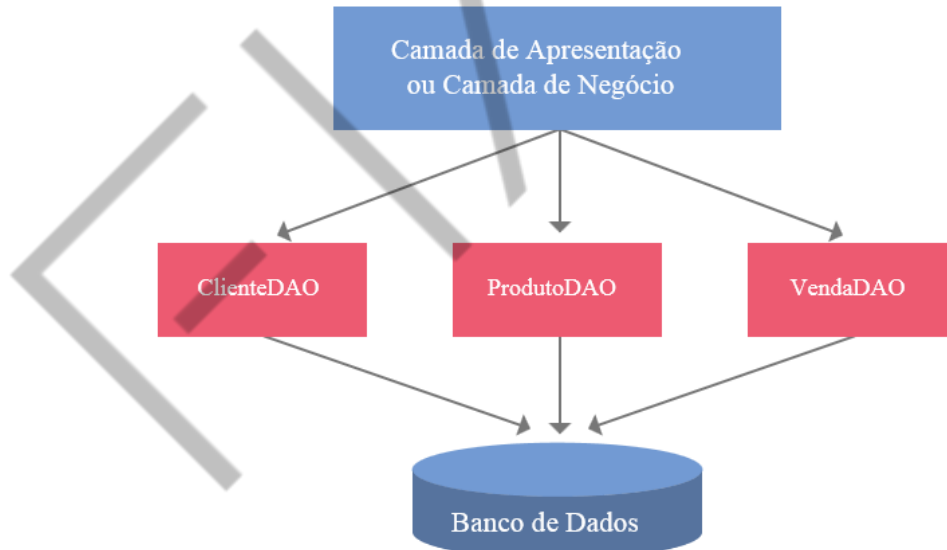


Figura 7.21 – Camada de acesso a dados da aplicação (DAO).  
Fonte: Fiap (2016)

O exemplo implementado acima já está utilizando o padrão DAO. Foi desenvolvida uma classe chamada **ColaboradorDAO**, que possui os métodos que realizam as operações no banco de dados.

O código abaixo apresenta o exemplo completo da classe ColaboradorDAO:

```
package br.com.fiap.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.List;

import br.com.fiap.bean.Colaborador;
import br.com.fiap.jdbc.EmpresaDBManager;

public class ColaboradorDAO {

    private Connection conexao;

    public void cadastrar(Colaborador colaborador) {
        PreparedStatement stmt = null;

        try {
            conexao = EmpresaDBManager.obterConexao();
            String sql = "INSERT INTO
TAB_COLABORADOR(CODIGO_COLABORADOR, NOME, EMAIL, SALARIO,
DATA_CONTRATACAO) VALUES (SQ_COLABORADOR.NEXTVAL, ?, ?, ?,
?)";

            stmt = conexao.prepareStatement(sql);
            stmt.setString(1, colaborador.getNome());
            stmt.setString(2, colaborador.getEmail());
            stmt.setDouble(3,
colaborador.getSalario());
```

```
        java.sql.Date data = new
java.sql.Date(colaborador.getDataContratacao().getTimeInMillis
());

        stmt.setDate(4, data);

        stmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            stmt.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

public List<Colaborador> listar() {
    //Cria uma lista de colaboradores
    List<Colaborador> lista = new
ArrayList<Colaborador>();
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conexao = EmpresaDBManager.obterConexao();
        stmt = conexao.prepareStatement("SELECT *
FROM TAB_COLABORADOR");
        rs = stmt.executeQuery();

        //Percorre todos os registros encontrados
        while (rs.next()) {
            int codigo =
rs.getInt("CODIGO_COLABORADOR");
```

```
String nome = rs.getString("NOME");
String email = rs.getString("EMAIL");
double salario =
rs.getDouble("SALARIO");
java.sql.Date data =
rs.getDate("DATA_CONTRATACAO");
Calendar dataContratacao =
Calendar.getInstance();

dataContratacao.setTimeInMillis(data.getTime());
//Cria um objeto Colaborador com as
informações encontradas
Colaborador colaborador = new
Colaborador(codigo, nome, email, salario, dataContratacao);
//Adiciona o colaborador na lista
lista.add(colaborador);
    }
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        stmt.close();
        rs.close();
        conexao.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
return lista;
}

public void atualizar(Colaborador colaborador) {
    PreparedStatement stmt = null;
```

```
        try {
            conexao = EmpresaDBManager.obterConexao();
            String sql = "UPDATE TAB_COLABORADOR SET
NOME = ?, EMAIL = ?, SALARIO = ?, DATA_CONTRATACAO = ? WHERE
CODIGO_COLABORADOR = ?";
            stmt = conexao.prepareStatement(sql);
            stmt.setString(1, colaborador.getNome());
            stmt.setString(2, colaborador.getEmail());
            stmt.setDouble(3,
colaborador.getSalario());
            java.sql.Date data = new
java.sql.Date(colaborador.getDataContratacao().getTimeInMillis
());

            stmt.setDate(4, data);
            stmt.setInt(5, colaborador.getCodigo());

            stmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                stmt.close();
                conexao.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    public void remover(int codigo) {
        PreparedStatement stmt = null;

        try {
            conexao = EmpresaDBManager.obterConexao();
```



```
String sql = "DELETE FROM TAB_COLABORADOR
WHERE CODIGO_COLABORADOR = ?";

stmt = conexao.prepareStatement(sql);
stmt.setInt(1, codigo);
stmt.executeUpdate();
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        stmt.close();
        conexao.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

}

public Colaborador buscarPorId(int codigoBusca) {
    Colaborador colaborador = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conexao = EmpresaDBManager.obterConexao();
        stmt = conexao.prepareStatement("SELECT *
FROM TAB_COLABORADOR WHERE CODIGO_COLABORADOR = ?");
        stmt.setInt(1, codigoBusca);
        rs = stmt.executeQuery();

        if (rs.next()) {
            int codigo =
rs.getInt("CODIGO_COLABORADOR");

            String nome = rs.getString("NOME");
            String email = rs.getString("EMAIL");
```

```
        double salario =  
rs.getDouble("SALARIO");  
        java.sql.Date data =  
rs.getDate("DATA_CONTRATACAO");  
        Calendar dataContratacao =  
Calendar.getInstance();  
  
        dataContratacao.setTimeInMillis(data.getTime());  
        colaborador = new Colaborador(codigo,  
nome, email, salario, dataContratacao);  
    }  
  
    } catch (SQLException e) {  
        e.printStackTrace();  
    } finally {  
        try {  
            stmt.close();  
            rs.close();  
            conexao.close();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
  
    return colaborador;  
}  
  
}
```

Para utilizar o DAO, basta instanciar a classe e invocar os seus métodos:

```
public static void main(String[] args) {  
  
    ColaboradorDAO dao = new ColaboradorDAO();  
  
    List<Colaborador> lista = dao.listar();  
  
}
```

Outra boa prática na implementação do padrão de projeto DAO é a utilização de interfaces. Dessa forma, a aplicação pode referenciar a interface ao invés da classe, diminuindo assim o acoplamento. Utilizar interface, permite a criação de diferentes implementações que podem ser trocadas sem a necessidade de grandes alterações no código da camada de negócio ou apresentação.

O exemplo abaixo apresenta a implementação do ColaboradorDAO utilizando uma interface e a implementação para o banco de dados Oracle:

Interface que define as funcionalidades do DAO:

```
package br.com.fiap.dao;  
  
import java.util.List;  
import br.com.fiap.bean.Colaborador;  
  
public interface ColaboradorDAO {  
  
    public void cadastrar(Colaborador colaborador);  
  
    public List<Colaborador> listar();  
  
    public void atualizar(Colaborador colaborador);  
  
    public void remover(int codigo);  

```

```
public Colaborador buscarPorId(int codigoBusca);  
}
```

Classe que implementa as funcionalidades definidas na interface:

```
package br.com.fiap.dao;  
  
import java.sql.Connection;  
import java.sql.PreparedStatement;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.util.ArrayList;  
import java.util.Calendar;  
import java.util.List;  
import br.com.fiap.bean.Colaborador;  
import br.com.fiap.jdbc.EmpresaDBManager;  
  
public class OracleColaboradorDAO implements  
ColaboradorDAO {  
  
    private Connection conexao;  
  
    public void cadastrar(Colaborador colaborador) {  
        PreparedStatement stmt = null;  
  
        try {  
            conexao = EmpresaDBManager.obterConexao();  
            String sql = "INSERT INTO  
TAB_COLABORADOR(CODIGO_COLABORADOR, NOME, EMAIL, SALARIO,  
DATA_CONTRATACAO) VALUES (SQ_COLABORADOR.NEXTVAL, ?, ?, ?,  
?)";  
  
            stmt = conexao.prepareStatement(sql);  
            stmt.setString(1, colaborador.getNome());  
            stmt.setString(2, colaborador.getEmail());  

```

```
        stmt.setDouble(3,
colaborador.getSalario());

        java.sql.Date data = new
java.sql.Date(colaborador.getDataContratacao().getTimeInMillis
());

        stmt.setDate(4, data);

        stmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            stmt.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

public List<Colaborador> listar() {
    //Cria uma lista de colaboradores
    List<Colaborador> lista = new
ArrayList<Colaborador>();

    PreparedStatement stmt = null;
    ResultSet rs = null;

    try {
        conexao = EmpresaDBManager.obterConexao();
        stmt = conexao.prepareStatement("SELECT *
FROM TAB_COLABORADOR");

        rs = stmt.executeQuery();

        //Percorre todos os registros encontrados
        while (rs.next()) {
```

```
        int codigo =  
rs.getInt("CODIGO_COLABORADOR");  
        String nome = rs.getString("NOME");  
        String email = rs.getString("EMAIL");  
        double salario =  
rs.getDouble("SALARIO");  
        java.sql.Date data =  
rs.getDate("DATA_CONTRATACAO");  
        Calendar dataContratacao =  
Calendar.getInstance();  
        dataContratacao.setTimeInMillis(data.getTime());  
        //Cria um objeto Colaborador com as  
informações encontradas  
        Colaborador colaborador = new  
Colaborador(codigo, nome, email, salario, dataContratacao);  
        //Adiciona o colaborador na lista  
        lista.add(colaborador);  
    }  
    } catch (SQLException e) {  
        e.printStackTrace();  
    } finally {  
        try {  
            stmt.close();  
            rs.close();  
            conexao.close();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
    }  
    return lista;  
}  
  
public void atualizar(Colaborador colaborador) {
```

```
PreparedStatement stmt = null;

    try {
        conexao = EmpresaDBManager.obterConexao();
        String sql = "UPDATE TAB_COLABORADOR SET
NOME = ?, EMAIL = ?, SALARIO = ?, DATA_CONTRATACAO = ? WHERE
CODIGO_COLABORADOR = ?";

        stmt = conexao.prepareStatement(sql);
        stmt.setString(1, colaborador.getNome());
        stmt.setString(2, colaborador.getEmail());
        stmt.setDouble(3,
colaborador.getSalario());
        java.sql.Date data = new
java.sql.Date(colaborador.getDataContratacao().getTimeInMillis
());

        stmt.setDate(4, data);
        stmt.setInt(5, colaborador.getCodigo());

        stmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            stmt.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

public void remover(int codigo){
    PreparedStatement stmt = null;

    try {
```

```
        conexao = EmpresaDBManager.obterConexao();
        String sql = "DELETE FROM TAB_COLABORADOR
WHERE CODIGO_COLABORADOR = ?";

        stmt = conexao.prepareStatement(sql);
        stmt.setInt(1, codigo);
        stmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        try {
            stmt.close();
            conexao.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

public Colaborador buscarPorId(int codigoBusca) {
    Colaborador colaborador = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conexao = EmpresaDBManager.obterConexao();
        stmt = conexao.prepareStatement("SELECT *
FROM TAB_COLABORADOR WHERE CODIGO_COLABORADOR = ?");

        stmt.setInt(1, codigoBusca);
        rs = stmt.executeQuery();

        if (rs.next()) {
            int codigo =
rs.getInt("CODIGO_COLABORADOR");

            String nome = rs.getString("NOME");
            String email = rs.getString("EMAIL");
```



```
        double salario =  
rs.getDouble("SALARIO");  
        java.sql.Date data =  
rs.getDate("DATA_CONTRATACAO");  
        Calendar dataContratacao =  
Calendar.getInstance();  
  
        dataContratacao.setTimeInMillis(data.getTime());  
        colaborador = new Colaborador(codigo,  
nome, email, salario, dataContratacao);  
    }  
    } catch (SQLException e) {  
        e.printStackTrace();  
    } finally {  
        try {  
            stmt.close();  
            rs.close();  
            conexao.close();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
    }  
    return colaborador;  
}  
}
```

#### 7.4.2 DAO Factory

Trata-se de uma fábrica de DAOs para um tipo específico de banco de dados. Esse padrão de projetos é recomendado quando não há previsão de mudar o banco de dados da aplicação, ou seja, se o sistema utiliza o banco de dados MySQL e não há previsão de trocar o banco de dados, esse padrão é o mais recomendado. Existem outros padrões mais indicados quando há a possibilidade da troca do banco de dados, que veremos a seguir.

A ideia do DAOFactory é centralizar a criação dos objetos DAOs, desta forma, se a aplicação precisar de um objeto ColaboradorDAO, o DAOFactory irá prover através de um método estático. Abaixo é apresentado um exemplo de implementação do DAOFactory que está em uma aplicação que possui dois DAOs:

Primeiro vamos implementar os DAOs do Colaborador e Cargo:

Interface CargoDAO e classe OracleCargoDAO, que implementa a interface:

```
public interface CargoDAO {

    public List<Cargo> listar();

    public void cadastrar(Cargo cargo);

}

public class OracleCargoDAO implements CargoDAO {

    @Override
    public List<Cargo> listar() {
        //Implementação
    }

    @Override
    public void cadastrar(Cargo cargo) {
        //Implementação
    }

}
```

Interface ColaboradorDAO e classe OracleColaboradorDAO, que implementa a interface:

```
public interface ColaboradorDAO {  
  
    public List<Colaborador> listar();  
  
    public void cadastrar(Colaborador colaborador);  
  
}  
  
public class OracleColaboradorDAO implements  
ColaboradorDAO {  
  
    public List<Colaborador> listar(){  
        //Implementação  
    }  
  
    public void cadastrar(Colaborador colaborador){  
        //Implementação  
    }  
  
}
```

Agora vamos implementar o DAOFactory, a fábrica que fornece as instâncias dos DAOs:

```
package br.com.fiap.factory;  
  
import br.com.fiap.dao.CargoDAO;  
import br.com.fiap.dao.ColaboradorDAO;  
import br.com.fiap.dao.OracleCargoDAO;  
import br.com.fiap.dao.OracleColaboradorDAO;  
  
public abstract class DAOFactory {  
  
    public static CargoDAO getCargoDAO() {  
        return new OracleCargoDAO();  
    }  
  
}
```

```
        public static ColaboradorDAO getColaboradorDAO() {  
            return new OracleColaboradorDAO();  
        }  
  
    }  
  
}
```

Note que a classe acima possui dois métodos estáticos que fornecem as instâncias dos DAOs. Para sua utilização, basta invocar esses métodos:

```
import java.util.List;  
  
import br.com.fiap.bean.Colaborador;  
import br.com.fiap.dao.ColaboradorDAO;  
import br.com.fiap.factory.DAOFactory;  
  
public class TesteDAOFactory {  
  
    public static void main(String[] args) {  
        ColaboradorDAO dao =  
        DAOFactory.getColaboradorDAO();  
        List<Colaborador> lista = dao.listar();  
    }  
  
}
```

### 7.4.3 Abstract Factory

Ao contrário do DAO Factory, este padrão de projetos é indicado quando é preciso suportar vários bancos de dados, pois permite criar implementações de DAOs para cada banco de dados e prover o DAO correto de acordo com a necessidade.

Exemplo: um sistema utiliza o banco de dados Oracle e SQL Server e é possível que no futuro passe a utilizar também DB2 para suportar um determinado módulo que foi adquirido pela empresa. Nesse cenário é justificada a utilização de uma estrutura

que permite suportar diversos tipos de bancos de dados e estar preparado para suportar novos tipos.

A implementação desse padrão de projetos começa pelos DAOs. Se a aplicação precisa suportar o Oracle e o SQL Server, por exemplo, é necessário criar uma interface que define as funcionalidades e implementá-la em duas classes, uma específica que acessa o banco de dados Oracle e outra que acessa o banco de dados SQL Server.

Depois de desenvolvida a estrutura do DAO, devemos implementar a Fábrica de DAOs que é uma classe abstrata (não pode ser instanciada) que deve possuir um método estático que retorna uma instância da Fábrica de DAO de acordo com o Banco de dados escolhido.

Exemplo: Primeiro definimos as interfaces do DAO, por exemplo, podemos ter as interfaces ColaboradorDAO e CargoDAO. Para cada interface, devemos implementar a classe que acessa o banco de dados Oracle e outro para o SQL. Assim, teremos as classes OracleColaboradorDAO e SQLColaboradorDAO para a interface ColaboradorDAO e OracleCargoDAO e SQLCargoDAO para a interface CargoDAO.

Agora precisamos desenvolver o Abstract DAO Factory que será responsável por fornecer a instância da fábrica de DAO, de acordo com o banco escolhido:

```
package br.com.fiap.factory;

import br.com.fiap.dao.CargoDAO;
import br.com.fiap.dao.ColaboradorDAO;

public abstract class DAOFactory {

    //Constantes que definem os tipos de Data Source
    suportados

    public static final int SQL_SERVER = 1;
    public static final int ORACLE = 2;

    //Atributos que armazenam as instancias de cada
    Fábrica
```

```
private static DAOFactory oracleDAOFactory;
private static DAOFactory sqlDAOFactory;

//Método que retorna a instancia de uma fábrica de
acordo com o banco

public static DAOFactory getDAOFactory(int banco){
    switch (banco) {
        case SQL_SERVER:
            if (sqlDAOFactory == null)
                sqlDAOFactory = new SQLDAOFactory();
            return sqlDAOFactory;
        case ORACLE:
            if (oracleDAOFactory == null)
                oracleDAOFactory = new
OracleDAOFactory();
            return oracleDAOFactory;
        default:
            return null;
    }
}

//Assinaturas de métodos que retornam a instancia do
DAO

public abstract CargoDAO getCargoDAO();
public abstract ColaboradorDAO getColaboradorDAO();

}
```

Foram definidas duas constantes, uma para cada banco de dados suportado pela aplicação. Dois atributos armazenam suas respectivas fábricas de DAO, uma para cada banco. O método **getDAOFactory()** recebe o valor de uma das constantes que representam o banco de dados para devolver a instância da fábrica correta.

Após a classe abstrata criada, é preciso criar duas classes para estender a DAO Factory, uma para cada tipo de fábrica. Assim, temos o SQLDAOFactory e o OracleDAOFactory. Nestas classes implementamos os métodos que retornam as instâncias dos DAOs, de acordo com seu banco de dados.

Para utilizar o DAO Factory, basta invocar o método que recupera a instância da fábrica, informando o banco de dados escolhido:

```
package br.com.fiap.teste;

import java.util.List;

import br.com.fiap.bean.Colaborador;
import br.com.fiap.dao.ColaboradorDAO;
import br.com.fiap.factory.DAOFactory;

public class TesteDAOFactory {

    public static void main(String[] args) {

        ColaboradorDAO dao =
DAOFactory.getDAOFactory.DAOFactory.ORACLE).getColaboradorDAO(
);

        List<Colaborador> lista = dao.listar();

    }

}
```

#### 7.4.4 Singleton

O objetivo desse padrão de projetos é gerar somente uma instância da classe. Esse padrão oferece um ponto de acesso global, no qual todos podem acessar a instância da classe através desse ponto de acesso. A classe Singleton deve gerenciar a própria instância e evitar que qualquer outra classe crie uma instância dela.

Assim, a classe Singleton deve possuir um construtor privado (private) a fim de evitar que outras classes a instancie. Essa classe também deve possuir um método estático que devolve a instância da própria classe Singleton. Esse método deve validar se já existe uma instância da classe, caso não exista, ela deve criar uma nova, caso

contrário, deve retornar à instância existente. Para determinar se existe ou não a instância da classe, é preciso criar um atributo estático para armazená-la, conforme o exemplo abaixo:

```
package br.com.fiap.singleton;

import java.sql.Connection;

public class ConnectionManager {

    //Atributo que armazena a única instância de
    ConnectionManager

    private static ConnectionManager instance;

    //Construtor privado
    private ConnectionManager() {}

    public static ConnectionManager getInstance() {
        if (instance == null) {
            instance = new ConnectionManager();
        }
        return instance;
    }

    public Connection getConnection() {
        //Implementação
    }

}
```

A classe **ConnectionManager** possui uma variável privada e estática que se autorreferencia, esse atributo que armazenará a única instância da classe **ConnectionManager**. O construtor possui o modificador de acesso **private**, dessa forma, nenhuma outra classe tem acesso ao construtor, não permitindo a sua instanciação.



Para acessar a instância da classe **ConnectionManager** é preciso utilizar o método **getInstance()**, esse método gerencia a existência da única instância de **ConnectionManager**; valida se o atributo **instance** está vazio, e caso esteja, é criada a instância de **ConnectionManager**, atribuída a variável **instance** e retornado em seguida. Caso o atributo **instance** já possua um objeto, o método retorna o próprio objeto existente.

Para utilizar a classe **ConnectionManager**, devemos utilizar o método **getInstance()**, conforme o exemplo abaixo:

```
public class OracleColaboradorDAO implements
ColaboradorDAO {

    public List<Colaborador> listar(){
        Connection conexao =
        ConnectionManager.getInstance().getConnection();
        //Implementação
    }

    public void cadastrar(Colaborador colaborador){
        Connection conexao =
        ConnectionManager.getInstance().getConnection();
        //Implementação
    }
}
```

Note que não é possível instanciar o **ConnectionManager** de outra maneira, a não ser utilizando o método estático **getInstance()**.

Dessa forma, finalizamos o acesso ao banco de dados utilizando a linguagem de programação Java. Vimos alguns dos principais padrões de projetos utilizados no mercado e focado na implementação da camada de acesso a dados de uma aplicação.

## REFERÊNCIAS

BARNES, David J. **Programação Orientada a Objetos com Java**: Uma introdução Prática Utilizando Blue J. São Paulo: Pearson, 2004.

CADENHEAD, Rogers; LEMAY, Laura. **Aprenda em 21 dias Java 2 Professional Reference**. 5.ed. Rio de Janeiro: Elsevier, 2003.

DEITEL, Paul; DEITEL Harvey. **Java Como Programar**. 8.ed. São Paulo: Pearson, 2010.

HORSTMANN, Cay; CORNELL, Gary. **Core Java**. Volume I Fundamentos. 8.ed. São Paulo: Pearson 2009.

SIERRA, Kathy; BATES, Bert. **Use a cabeça! Java**. Rio de Janeiro: Alta Books, 2010.