

Simulando um Banco de Dados com uma Lista `static` em Java

Em uma aplicação Java, normalmente os dados são armazenados em um banco de dados como PostgreSQL, MySQL ou Oracle. Entretanto, para fins didáticos ou para criar protótipos, podemos simular um banco de dados utilizando uma coleção em memória.

Uma das formas mais simples é utilizar uma `List` associada à classe por meio do modificador `static`.

A ideia é criar uma classe responsável por armazenar os objetos e utilizar uma lista estática para manter os registros enquanto a aplicação estiver em execução.

1. Criando a classe `Cliente`

Inicialmente, vamos criar uma classe que representa os registros que serão armazenados.

```
public class Cliente {

    private Long id;
    private String nome;
    private String email;

    public Cliente(Long id, String nome, String email) {
        this.id = id;
        this.nome = nome;
        this.email = email;
    }

    public Long getId() {
        return id;
    }

    public String getNome() {
        return nome;
    }

    public String getEmail() {
        return email;
    }

    @Override
    public String toString() {
        return "Cliente{" +
            "id=" + id +
            ", nome='" + nome + '\'' +
```

```

        ", email='" + email + '\\'' +
        '}';
    }
}

```

A classe `Cliente` representa uma tabela que poderíamos ter em um banco de dados.

Podemos imaginar:

Tabela CLIENTE

ID	NOME	EMAIL
1	João	joao@email.com
2	Maria	maria@email.com
3	Carlos	carlos@email.com

Cada objeto `Cliente` representa um registro dessa tabela.

2. Criando um banco de dados em memória

Agora podemos criar uma classe chamada `BancoDeDados`.

```

import java.util.ArrayList;
import java.util.List;

public class BancoDeDados {

    private static final List<Cliente> clientes =
        new ArrayList<>();

}

```

Observe a declaração:

```

private static final List<Cliente> clientes =
    new ArrayList<>();

```

Temos três conceitos importantes:

`private`

A lista não pode ser acessada diretamente por qualquer classe.

`static`

Existe apenas **uma lista compartilhada pela classe**.

`final`

A referência da lista não poderá apontar para outro objeto `List`.

Isso não significa que a lista seja imutável.

Podemos continuar adicionando ou removendo elementos:

```
clientes.add(cliente);
clientes.remove(cliente);
```

O que não podemos fazer é:

```
clientes = new ArrayList<>();
```

3. Inserindo registros

Podemos criar um método para cadastrar clientes:

```
public static void adicionarCliente(Cliente cliente) {
    clientes.add(cliente);
}
```

A classe completa ficará:

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

public class BancoDeDados {

    private static final List<Cliente> clientes =
        new ArrayList<>();

    public static void adicionarCliente(Cliente cliente) {
        clientes.add(cliente);
    }
}
```

Agora podemos inserir clientes:

```
BancoDeDados.adicionarCliente(  
    new Cliente(1L, "João", "joao@email.com")  
);  
  
BancoDeDados.adicionarCliente(  
    new Cliente(2L, "Maria", "maria@email.com")  
);  
  
BancoDeDados.adicionarCliente(  
    new Cliente(3L, "Carlos", "carlos@email.com")  
);
```

Os objetos ficam armazenados na lista estática.

4. Consultando os registros

Podemos criar um método para retornar todos os clientes:

```
public static List<Cliente> listarClientes() {  
  
    return clientes;  
}
```

Podemos então utilizar:

```
List<Cliente> lista =  
    BancoDeDados.listarClientes();  
  
for (Cliente cliente : lista) {  
  
    System.out.println(cliente);  
}
```

Resultado:

```
Cliente{id=1, nome='João', email='joao@email.com'}  
  
Cliente{id=2, nome='Maria', email='maria@email.com'}  
  
Cliente{id=3, nome='Carlos', email='carlos@email.com'}
```

5. Melhorando o método de consulta

É recomendável evitar retornar diretamente a lista interna.

Podemos utilizar:

```
return new ArrayList<>(clientes);
```

Assim:

```
public static List<Cliente> listarClientes() {  
    return new ArrayList<>(clientes);  
}
```

O método retorna uma cópia da lista.

A classe fica:

```
import java.util.ArrayList;  
import java.util.List;  
  
public class BancoDeDados {  
    private static final List<Cliente> clientes =  
        new ArrayList<>();  
  
    public static void adicionarCliente(Cliente cliente) {  
        clientes.add(cliente);  
    }  
  
    public static List<Cliente> listarClientes() {  
        return new ArrayList<>(clientes);  
    }  
}
```

6. Buscando um cliente pelo ID

Podemos implementar uma operação semelhante a um:

```
SELECT * FROM CLIENTE WHERE ID = 2;
```

Em Java:

```
public static Cliente buscarPorId(Long id) {  
  
    for (Cliente cliente : clientes) {  
  
        if (cliente.getId().equals(id)) {  
            return cliente;  
        }  
    }  
  
    return null;  
}
```

Podemos utilizar:

```
Cliente cliente =  
    BancoDeDados.buscarPorId(2L);  
  
System.out.println(cliente);
```

Resultado:

```
Cliente{id=2, nome='Maria', email='maria@email.com'}
```

7. Utilizando Stream

A mesma operação pode ser realizada utilizando Stream API:

```
public static Cliente buscarPorId(Long id) {  
  
    return clientes.stream()  
        .filter(cliente -> cliente.getId().equals(id))  
        .findFirst()  
        .orElse(null);  
}
```

Essa implementação é bastante comum em aplicações modernas Java.

8. Removendo um registro

Podemos criar uma operação semelhante a:

```
DELETE FROM CLIENTE WHERE ID = 2;
```

Em Java:

```
public static boolean remover(Long id) {  
  
    return clientes.removeIf(  
        cliente -> cliente.getId().equals(id)  
    );  
}
```

Podemos utilizar:

```
boolean removido =  
    BancoDeDados.remover(2L);  
  
System.out.println(removido);
```

Resultado:

```
true
```

O cliente com ID `2` será removido da lista.

9. Atualizando um registro

Também podemos simular:

```
UPDATE CLIENTE  
SET NOME = 'Maria Silva'  
WHERE ID = 2;
```

Primeiro precisamos permitir alteração dos dados na classe `Cliente`:

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

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

Depois podemos criar o método de atualização:

```
public static boolean atualizar(Cliente clienteAtualizado) {

    for (Cliente cliente : clientes) {

        if (cliente.getId().equals(clienteAtualizado.getId())) {

            cliente.setNome(clienteAtualizado.getNome());
            cliente.setEmail(clienteAtualizado.getEmail());

            return true;
        }
    }

    return false;
}
```

Agora podemos fazer:

```
Cliente cliente =
    new Cliente(
        2L,
        "Maria Silva",
        "maria.silva@email.com"
    );

BancoDeDados.atualizar(cliente);
```

10. Criando um CRUD completo

Nesse momento temos as quatro operações básicas de um CRUD:

Operação	Método Java	SQL equivalente
Create	<code>adicionarCliente()</code>	<code>INSERT</code>
Read	<code>listarClientes()</code> / <code>buscarPorId()</code>	<code>SELECT</code>
Update	<code>atualizar()</code>	<code>UPDATE</code>
Delete	<code>remover()</code>	<code>DELETE</code>

Nossa classe pode ficar assim:

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

public class BancoDeDados {

    private static final List<Cliente> clientes =
        new ArrayList<>();

    public static void adicionarCliente(Cliente cliente) {

        clientes.add(cliente);
    }

    public static List<Cliente> listarClientes() {

        return new ArrayList<>(clientes);
    }

    public static Cliente buscarPorId(Long id) {

        return clientes.stream()
            .filter(cliente ->
                cliente.getId().equals(id))
            .findFirst()
            .orElse(null);
    }

    public static boolean atualizar(
        Cliente clienteAtualizado) {

        for (Cliente cliente : clientes) {

            if (cliente.getId()
                .equals(clienteAtualizado.getId())) {

                cliente.setNome(
                    clienteAtualizado.getNome());

                cliente.setEmail(
                    clienteAtualizado.getEmail());

                return true;
            }
        }

        return false;
    }
}
```

```
public static boolean remover(Long id) {  
  
    return clientes.removeIf(  
        cliente ->  
            cliente.getId().equals(id));  
    }  
}
```

11. Utilizando o banco de dados em memória

Podemos criar uma classe `Main` para testar nossa aplicação:

```
public class Main {  
  
    public static void main(String[] args) {  
  
        // CREATE  
  
        BancoDeDados.adicionarCliente(  
            new Cliente(  
                1L,  
                "João",  
                "joao@email.com"  
            )  
        );  
  
        BancoDeDados.adicionarCliente(  
            new Cliente(  
                2L,  
                "Maria",  
                "maria@email.com"  
            )  
        );  
  
        BancoDeDados.adicionarCliente(  
            new Cliente(  
                3L,  
                "Carlos",  
                "carlos@email.com"  
            )  
        );  
  
        // READ  
  
        System.out.println("CLIENTES:");
```

```

        for (Cliente cliente :
            BancoDeDados.listarClientes()) {

            System.out.println(cliente);
        }

        // UPDATE

        BancoDeDados.atualizar(
            new Cliente(
                2L,
                "Maria Silva",
                "maria.silva@email.com"
            )
        );

        // DELETE

        BancoDeDados.remover(3L);

        System.out.println();
        System.out.println("APÓS ALTERAÇÕES:");

        for (Cliente cliente :
            BancoDeDados.listarClientes()) {

            System.out.println(cliente);
        }
    }
}

```

12. Por que utilizar `static` nesse exemplo?

A principal razão é que queremos que exista **uma única lista representando o nosso banco de dados em memória**.

Imagine que não utilizássemos `static`:

```

public class BancoDeDados {

    private final List<Cliente> clientes =
        new ArrayList<>();
}

```

Nesse caso, cada objeto `BancoDeDados` teria sua própria lista.

Por exemplo:

```
BancoDeDados banco1 = new BancoDeDados();  
BancoDeDados banco2 = new BancoDeDados();
```

Teríamos:

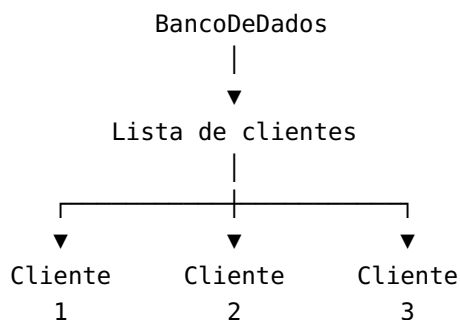
```
banco1  
└─ lista de clientes  
  
banco2  
└─ outra lista de clientes
```

Isso não representa bem a ideia de um banco de dados único.

Com `static`:

```
private static final List<Cliente> clientes =  
    new ArrayList<>();
```

temos:



Todos os métodos estáticos da classe trabalham sobre a mesma lista.

13. Importante: isso não é um banco de dados real

Apesar de ser útil para aprendizado, essa solução possui uma grande limitação:

Os dados estão armazenados apenas na memória da aplicação.

Se o programa for encerrado:

```
Aplicação executando
    ↓
Lista em memória
    ↓
Clientes cadastrados
```

Quando a aplicação for encerrada:

```
Aplicação encerrada
    ↓
Memória liberada
    ↓
Dados perdidos
```

Portanto, essa abordagem **não substitui um banco de dados real**.

Em uma aplicação de produção, poderíamos utilizar:

```
Java
    ↓
Spring Boot
    ↓
Spring Data JPA
    ↓
Hibernate
    ↓
PostgreSQL / MySQL / Oracle
```

14. Vantagem para fins didáticos

Apesar da limitação, utilizar uma `List` `static` é excelente para ensinar os primeiros conceitos de persistência de dados.

O aluno consegue compreender a relação:

```
Objeto Java
    ↓
Registro
    ↓
List<Objeto>
```

↓
"Banco de dados" em memória

Antes de aprender:

JPA
Hibernate
Repository
Entity
SQL
PostgreSQL

é possível compreender primeiro a lógica do CRUD utilizando apenas Java.

Conclusão

Uma `List` `static` pode ser utilizada para criar um **banco de dados simulado em memória**.

O conceito principal é:

```
private static final List<Cliente> clientes =  
    new ArrayList<>();
```

O `static` garante que exista uma única lista compartilhada pela classe `BancoDeDados`.

A partir dessa lista podemos implementar operações semelhantes às de um banco de dados real:

```
CREATE → adicionarCliente()  
READ  → listarClientes() / buscarPorId()  
UPDATE → atualizar()  
DELETE → remover()
```

Essa abordagem é particularmente interessante em aulas de Java porque permite demonstrar **classes**, **objetos**, **encapsulamento**, `static`, `List`, `ArrayList`, **Stream API** e **CRUD** sem introduzir imediatamente a complexidade de um banco de dados externo.

Posteriormente, a mesma ideia pode ser evoluída para uma aplicação com **Spring Boot + Spring Data JPA + PostgreSQL**, substituindo a lista em memória por um banco de dados persistente.