

Escopo de Classe, Controle de Acesso e Construtores

1. Introdução

Na programação orientada a objetos, uma classe representa uma estrutura que reúne **dados e comportamentos** relacionados a uma determinada entidade.

Em Java, ao criar uma classe, é importante compreender três conceitos fundamentais:

- **Escopo de classe:** determina onde atributos e métodos podem ser utilizados.
- **Controle de acesso:** define quem pode acessar os membros de uma classe.
- **Construtores:** permitem inicializar objetos no momento de sua criação.

Esses conceitos estão diretamente relacionados aos princípios de **encapsulamento, organização do código, segurança e manutenção de sistemas**.

2. Escopo de Classe

O escopo determina a região do programa na qual um elemento pode ser utilizado.

Dentro de uma classe Java podemos declarar, entre outros elementos:

- atributos;
- métodos;
- construtores;
- constantes;
- classes internas.

Considere a classe:

```
public class Produto {  
  
    private String nome;  
    private double preco;  
  
    public void exibirDados() {  
        System.out.println(nome);  
        System.out.println(preco);  
    }  
}
```

Os atributos `nome` e `preco` pertencem ao **escopo da classe**.

Eles podem ser utilizados diretamente pelos métodos da própria classe.

Por exemplo:

```
public void aplicarDesconto(double percentual) {  
    preco = preco - (preco * percentual / 100);  
}
```

O método `aplicarDesconto()` consegue acessar diretamente o atributo `preco`, pois ambos pertencem ao mesmo objeto e ao escopo da classe.

2.1 Atributos de instância

Um atributo declarado dentro da classe, mas fora dos métodos, normalmente representa uma característica do objeto.

```
public class Pessoa {  
  
    String nome;  
    int idade;  
}
```

Cada objeto possui sua própria cópia desses atributos.

```
Pessoa pessoa1 = new Pessoa();  
pessoa1.nome = "Carlos";  
pessoa1.idade = 30;  
  
Pessoa pessoa2 = new Pessoa();  
pessoa2.nome = "Maria";  
pessoa2.idade = 25;
```

Temos:

```
pessoa1  
├─ nome = Carlos  
└─ idade = 30  
  
pessoa2  
├─ nome = Maria  
└─ idade = 25
```

Assim, alterar `pessoa1.nome` não modifica `pessoa2.nome`.

3. Escopo de Variáveis

Além do escopo da classe, Java possui variáveis com escopo mais restrito.

Por exemplo:

```
public class Calculadora {  
  
    private int resultado;  
  
    public void somar(int a, int b) {  
  
        int soma = a + b;  
  
        resultado = soma;  
    }  
}
```

Nesse exemplo:

- `resultado` possui **escopo de classe**;
- `a` e `b` são parâmetros do método;
- `soma` possui **escopo local**, existindo somente dentro do método `somar()`.

Portanto, o código abaixo provocaria erro:

```
public void exibir() {  
  
    System.out.println(soma);  
}
```

A variável `soma` não existe nesse escopo.

4. Palavra-chave `this`

A palavra-chave `this` representa a **instância atual da classe**.

Ela é muito utilizada quando existe conflito entre o nome de um atributo e o nome de um parâmetro.

Exemplo:

```
public class Cliente {  
  
    private String nome;  
    private String email;
```

```
public void cadastrar(String nome, String email) {  
  
    this.nome = nome;  
    this.email = email;  
  
}
```

Observe:

```
this.nome = nome;
```

Nesse caso:

- `this.nome` representa o atributo da classe;
- `nome` representa o parâmetro do método.

O mesmo ocorre com:

```
this.email = email;
```

Essa é uma das aplicações mais comuns da palavra-chave `this` em Java.

5. Controle de Acesso

O controle de acesso determina quais partes de uma aplicação podem acessar atributos, métodos e outros membros de uma classe.

Java possui quatro níveis principais de acesso:

Modificador	Própria classe	Mesmo pacote	Subclasses	Qualquer classe
<code>private</code>	Sim	Não	Não diretamente	Não
padrão	Sim	Sim	Sim*	Não
<code>protected</code>	Sim	Sim	Sim	Não diretamente**
<code>public</code>	Sim	Sim	Sim	Sim

* O acesso padrão depende do relacionamento de pacote.

** Uma subclasse em outro pacote possui regras específicas de acesso ao membro `protected`.

Os principais modificadores são:

```
private
protected
public
```

Além deles, existe o acesso **padrão**, quando nenhum modificador é informado.

6. Modificador `private`

O modificador `private` restringe o acesso ao próprio contexto da classe.

Por exemplo:

```
public class Conta {

    private double saldo;

    public void depositar(double valor) {
        saldo += valor;
    }
}
```

Outra classe não pode fazer:

```
Conta conta = new Conta();

conta.saldo = 1000;
```

Isso provocará erro de compilação porque `saldo` é `private`.

A alteração deve ocorrer por meio de um comportamento fornecido pela classe:

```
conta.depositar(1000);
```

Esse mecanismo contribui para o **encapsulamento**.

7. Modificador `public`

Um membro declarado como `public` pode ser acessado por outras classes.

Exemplo:

```
public class Calculadora {  
  
    public int somar(int a, int b) {  
        return a + b;  
    }  
}
```

Outra classe pode utilizar o método:

```
Calculadora calculadora = new Calculadora();  
  
int resultado = calculadora.somar(10, 20);  
  
System.out.println(resultado);
```

Resultado:

```
30
```

Métodos públicos normalmente representam operações que fazem parte da **interface pública** de uma classe.

8. Modificador `protected`

O modificador `protected` permite acesso:

- dentro da própria classe;
- por classes do mesmo pacote;
- por subclasses, respeitando as regras de herança e pacote.

Exemplo:

```
public class Animal {  
  
    protected String nome;  
  
    public void emitirSom() {  
        System.out.println("Som do animal");  
    }  
}
```

Uma classe filha pode utilizar esse atributo:

```
public class Cachorro extends Animal {  
  
    public void apresentar() {  
        System.out.println("Nome: " + nome);  
    }  
}
```

Nesse caso, `Cachorro` herda de `Animal` e consegue acessar `nome`.

9. Acesso Padrão

Quando nenhum modificador de acesso é informado, Java utiliza o chamado **package-private**, ou acesso padrão.

Exemplo:

```
class Produto {  
  
    String nome;  
    double preco;  
}
```

Nesse caso, os membros podem ser acessados por classes que pertencem ao **mesmo pacote**.

Esse recurso pode ser útil para componentes que precisam colaborar internamente, mas não devem fazer parte da API pública da aplicação.

10. Encapsulamento

O controle de acesso está diretamente relacionado ao conceito de **encapsulamento**.

Encapsular significa proteger os dados internos de um objeto e controlar como eles podem ser manipulados.

Um exemplo clássico:

```
public class Conta {  
  
    private double saldo;  
  
    public void depositar(double valor) {  
  
        if (valor > 0) {
```

```
        saldo += valor;
    }
}

public double consultarSaldo() {
    return saldo;
}
}
```

O atributo:

```
private double saldo;
```

não pode ser alterado diretamente por outras classes.

A classe controla a operação:

```
if (valor > 0)
```

Dessa forma, evita-se uma situação inválida, como:

```
saldo = -5000;
```

O objeto passa a ser responsável por proteger seu próprio estado.

11. Métodos Getters e Setters

Uma abordagem bastante comum para acessar atributos privados é utilizar métodos chamados **getters** e **setters**.

Exemplo:

```
public class Pessoa {

    private String nome;
    private int idade;

    public String getNome() {
        return nome;
    }

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

```
public int getIdade() {  
    return idade;  
}  
  
public void setIdade(int idade) {  
    this.idade = idade;  
}  
}
```

Utilização:

```
Pessoa pessoa = new Pessoa();  
  
pessoa.setNome("João");  
pessoa.setIdade(35);  
  
System.out.println(pessoa.getNome());  
System.out.println(pessoa.getIdade());
```

Entretanto, é importante destacar que **encapsulamento não significa simplesmente criar getter e setter para todos os atributos**.

Um setter pode permitir alterações que deveriam ser controladas por uma regra de negócio.

Por exemplo, em vez de:

```
conta.setSaldo(-1000);
```

é melhor fornecer operações específicas:

```
conta.depositar(500);  
conta.sacar(200);
```

Assim, a classe controla as regras relacionadas ao seu estado.

12. Construtores

Um **construtor** é um recurso utilizado para inicializar um objeto durante sua criação.

Ele possui algumas características importantes:

- possui o mesmo nome da classe;
- não possui tipo de retorno;

- é executado quando utilizamos `new`;
- pode receber parâmetros;
- pode existir em diferentes versões por sobrecarga.

Exemplo:

```
public class Produto {  
  
    private String nome;  
    private double preco;  
  
    public Produto() {  
        nome = "Produto não informado";  
        preco = 0;  
    }  
}
```

Ao executar:

```
Produto produto = new Produto();
```

o construtor é executado automaticamente.

13. Construtor com Parâmetros

Podemos utilizar parâmetros para criar objetos já inicializados.

```
public class Produto {  
  
    private String nome;  
    private double preco;  
  
    public Produto(String nome, double preco) {  
        this.nome = nome;  
        this.preco = preco;  
    }  
}
```

Agora podemos fazer:

```
Produto produto = new Produto("Notebook", 3500.00);
```

Após a criação, o objeto terá:

```
nome = Notebook  
preco = 3500.00
```

Essa abordagem é muito utilizada quando determinado objeto precisa nascer com informações obrigatórias.

14. Construtores e Validação

Uma aplicação interessante dos construtores é garantir que um objeto seja criado em um estado válido.

Por exemplo:

```
public class Produto {  
  
    private String nome;  
    private double preco;  
  
    public Produto(String nome, double preco) {  
  
        if (nome == null || nome.isBlank()) {  
            throw new IllegalArgumentException("Nome inválido");  
        }  
  
        if (preco < 0) {  
            throw new IllegalArgumentException("Preço inválido");  
        }  
  
        this.nome = nome;  
        this.preco = preco;  
    }  
}
```

Agora:

```
Produto produto = new Produto("Teclado", 150.00);
```

é válido.

Porém:

```
Produto produto = new Produto("", -100);
```

provocará uma exceção.

O construtor funciona, portanto, como uma porta de entrada para a criação de um objeto consistente.

15. Sobrecarga de Construtores

Uma classe pode possuir mais de um construtor, desde que eles possuam diferentes listas de parâmetros.

Exemplo:

```
public class Pessoa {  
  
    private String nome;  
    private int idade;  
  
    public Pessoa() {  
        nome = "Não informado";  
        idade = 0;  
    }  
  
    public Pessoa(String nome) {  
        this.nome = nome;  
        idade = 0;  
    }  
  
    public Pessoa(String nome, int idade) {  
        this.nome = nome;  
        this.idade = idade;  
    }  
}
```

Podemos criar objetos de diferentes formas:

```
Pessoa p1 = new Pessoa();  
  
Pessoa p2 = new Pessoa("Maria");  
  
Pessoa p3 = new Pessoa("Carlos", 30);
```

Essa técnica é chamada de **sobrecarga de construtores**.

16. Utilizando `this()` entre construtores

É possível fazer um construtor chamar outro construtor da mesma classe utilizando `this()`.

Exemplo:

```
public class Pessoa {  
  
    private String nome;  
    private int idade;  
  
    public Pessoa() {  
        this("Não informado", 0);  
    }  
  
    public Pessoa(String nome) {  
        this(nome, 0);  
    }  
  
    public Pessoa(String nome, int idade) {  
        this.nome = nome;  
        this.idade = idade;  
    }  
}
```

Nesse exemplo, existe uma centralização da inicialização.

O construtor principal é:

```
public Pessoa(String nome, int idade)
```

Os outros construtores delegam a ele.

Isso reduz duplicação de código.

17. Construtor Padrão

É importante diferenciar **construtor padrão** de um construtor sem parâmetros escrito pelo programador.

Quando uma classe não possui nenhum construtor declarado, o compilador fornece automaticamente um construtor sem parâmetros.

Exemplo:

```
public class Produto {
```

```
        private String nome;  
    }
```

Podemos fazer:

```
Produto produto = new Produto();
```

Porém, se declararmos um construtor:

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

o construtor sem parâmetros não será mais criado automaticamente.

Assim:

```
Produto produto = new Produto();
```

passará a gerar erro, pois não existe mais um construtor sem argumentos.

Se quisermos os dois, precisamos declarar explicitamente:

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

18. Aplicação Prática: Sistema de Funcionários

Podemos combinar escopo, controle de acesso e construtores em uma classe de aplicação.

```
public class Funcionario {  
  
    private String nome;  
    private double salario;  
  
    public Funcionario(String nome, double salario) {  
  
        if (nome == null || nome.isBlank()) {
```

```

        throw new IllegalArgumentException("Nome inválido");
    }

    if (salario < 0) {
        throw new IllegalArgumentException("Salário inválido");
    }

    this.nome = nome;
    this.salario = salario;
}

public String getNome() {
    return nome;
}

public double getSalario() {
    return salario;
}

public void aumentarSalario(double percentual) {

    if (percentual > 0) {
        salario += salario * percentual / 100;
    }
}
}

```

Podemos utilizar a classe:

```

public class Main {

    public static void main(String[] args) {

        Funcionario funcionario =
            new Funcionario("Carlos", 3000);

        System.out.println(funcionario.getNome());
        System.out.println(funcionario.getSalario());

        funcionario.aumentarSalario(10);

        System.out.println(funcionario.getSalario());
    }
}

```

Resultado:

```
Carlos
3000.0
3300.0
```

Nesse exemplo estão presentes vários conceitos:

- `nome` e `salario` possuem escopo de classe;
- os atributos são `private`;
- o acesso ocorre por métodos públicos;
- o construtor inicializa o objeto;
- o construtor realiza validações;
- `this` diferencia atributos de parâmetros;
- o método `aumentarSalario()` controla uma alteração no estado do objeto.

19. Escopo, Acesso e Construtores trabalhando juntos

Esses três conceitos não devem ser estudados de forma isolada.

Considere o seguinte modelo:

CLASSE FUNCIONARIO

```
- nome
- salario

+ Funcionario(...)
+ getNome()
+ getSalario()
+ aumentarSalario()
```

O símbolo `-` representa atributos privados e `+` representa membros públicos.

O objeto é criado por meio do construtor:

```
Funcionario f =
    new Funcionario("Ana", 4000);
```

Os atributos ficam protegidos:

```
private String nome;  
private double salario;
```

E as operações são disponibilizadas por métodos públicos:

```
public double getSalario()
```

Esse modelo favorece o encapsulamento e torna a classe mais segura e organizada.

20. Boas Práticas

Ao desenvolver classes Java, algumas práticas são recomendadas:

1. Proteja os atributos

Sempre que possível, mantenha os atributos como:

```
private
```

2. Exponha comportamentos

Prefira métodos que representem operações do domínio:

```
depositar()  
sacar()  
aumentarSalario()  
cancelar()  
ativar()
```

em vez de permitir alterações indiscriminadas nos atributos.

3. Valide dados importantes

Construtores e métodos podem impedir que objetos entrem em estados inválidos.

4. Utilize `this` de forma clara

Principalmente quando parâmetros possuem o mesmo nome dos atributos:

```
this.nome = nome;
```

5. Evite duplicação entre construtores

Utilize:

```
this(...)
```

para centralizar a inicialização.

6. Não crie getters e setters automaticamente para tudo

Pergunte se o atributo realmente precisa ser exposto ou alterado externamente.

21. Conclusão

Escopo de classe, controle de acesso e construtores são conceitos fundamentais para o desenvolvimento de aplicações orientadas a objetos em Java.

O escopo determina **onde os elementos podem ser utilizados**. Os modificadores de acesso determinam **quem pode acessá-los**. Já os construtores permitem definir **como um objeto deve ser inicializado**.

Quando esses recursos são utilizados corretamente, podemos criar classes que:

- protegem seus dados;
- controlam suas próprias regras;
- evitam estados inválidos;
- facilitam a manutenção;
- promovem reutilização;
- tornam o código mais organizado.

Em aplicações reais, esses conceitos aparecem constantemente em classes como:

```
Cliente  
Produto  
Pedido  
Funcionario  
Usuario  
Conta  
Veiculo  
Endereco  
Pagamento
```

Por isso, compreender escopo, encapsulamento, controle de acesso e construtores é um passo essencial para avançar no estudo da **Programação Orientada a Objetos em Java**.