

Atributos e Métodos `static`

Na programação orientada a objetos, os atributos e métodos normalmente pertencem a uma **instância de uma classe**, ou seja, a um objeto criado a partir dessa classe.

Entretanto, em determinadas situações, queremos que um atributo ou método pertença à **própria classe**, e não a cada objeto individualmente. Para isso, Java oferece o modificador `static`.

Um membro declarado como `static` é compartilhado por todas as instâncias da classe e pode ser acessado sem a necessidade de criar um objeto.

1. O que significa `static`?

A palavra-chave `static` indica que um atributo ou método pertence à **classe**.

Por exemplo:

```
public class Pessoa {  
  
    String nome;  
    static int quantidadePessoas;  
}
```

Nesse exemplo:

- `nome` é um atributo de **instância**;
- `quantidadePessoas` é um atributo **estático**;
- cada objeto possui seu próprio `nome`;
- todos os objetos compartilham a mesma `quantidadePessoas`.

Podemos representar essa diferença da seguinte forma:

```
Classe Pessoa  
├── atributo de instância: nome  
│   └── cada objeto possui seu próprio valor  
└── atributo static: quantidadePessoas  
    └── um único valor compartilhado
```

2. Atributos `static`

Um atributo `static` possui apenas uma cópia associada à classe, independentemente da quantidade de objetos criados.

Considere:

```
public class Pessoa {  
  
    String nome;  
    static int quantidadePessoas = 0;  
  
    public Pessoa(String nome) {  
        this.nome = nome;  
        quantidadePessoas++;  
    }  
}
```

Podemos criar vários objetos:

```
public class Main {  
  
    public static void main(String[] args) {  
  
        Pessoa p1 = new Pessoa("João");  
        Pessoa p2 = new Pessoa("Maria");  
        Pessoa p3 = new Pessoa("Carlos");  
  
        System.out.println(Pessoa.quantidadePessoas);  
    }  
}
```

Resultado:

3

Observe que não existe uma `quantidadePessoas` para cada objeto.

Existe apenas uma variável:

`Pessoa.quantidadePessoas`

Ela é compartilhada por `p1`, `p2` e `p3`.

3. Acesso a atributos `static`

Embora Java permita acessar um atributo `static` através de um objeto, essa não é a forma recomendada.

Por exemplo:

```
Pessoa p1 = new Pessoa("João");  
  
System.out.println(p1.quantidadePessoas);
```

Apesar de funcionar, o ideal é acessar o atributo através do nome da classe:

```
System.out.println(Pessoa.quantidadePessoas);
```

Essa forma deixa claro que o atributo pertence à classe e não ao objeto.

4. Constantes com `static final`

Uma das aplicações mais comuns de `static` é a criação de **constantas**.

Normalmente utilizamos:

```
static final
```

Por exemplo:

```
public class Matematica {  
  
    public static final double PI = 3.14159265359;  
}
```

A constante pode ser utilizada diretamente:

```
public class Main {  
  
    public static void main(String[] args) {  
  
        System.out.println(Matematica.PI);  
    }  
}
```

O `static` indica que a constante pertence à classe.

O `final` indica que seu valor não pode ser alterado depois de inicializado.

É comum utilizar nomes de constantes em letras maiúsculas:

```
public static final int IDADE_MINIMA = 18;
public static final double TAXA_JUROS = 0.05;
public static final String NOME_SISTEMA = "Sistema Financeiro";
```

5. Métodos `static`

Assim como os atributos, os métodos também podem ser declarados como `static`.

Um método `static` pertence à classe e pode ser chamado sem a criação de um objeto.

Por exemplo:

```
public class Calculadora {

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

Podemos utilizar o método diretamente:

```
public class Main {

    public static void main(String[] args) {

        int resultado = Calculadora.somar(10, 20);

        System.out.println(resultado);
    }
}
```

Resultado:

30

Não precisamos fazer:

```
Calculadora calculadora = new Calculadora();
```

O método pode ser chamado diretamente através da classe:

```
Calculadora.somar(10, 20);
```

6. Quando utilizar métodos `static`?

Métodos `static` são apropriados quando o comportamento do método **não depende dos dados de um objeto específico**.

Por exemplo, uma calculadora pode possuir operações matemáticas:

```
public class Calculadora {  
  
    public static int somar(int a, int b) {  
        return a + b;  
    }  
  
    public static int subtrair(int a, int b) {  
        return a - b;  
    }  
  
    public static int multiplicar(int a, int b) {  
        return a * b;  
    }  
  
    public static double dividir(double a, double b) {  
        return a / b;  
    }  
}
```

Utilização:

```
public class Main {  
  
    public static void main(String[] args) {  
  
        System.out.println(Calculadora.somar(10, 5));  
        System.out.println(Calculadora.subtrair(10, 5));  
        System.out.println(Calculadora.multiplicar(10, 5));  
        System.out.println(Calculadora.dividir(10, 5));  
    }  
}
```

```
}  
}
```

Resultado:

```
15  
5  
50  
2.0
```

Nesse caso, não existe necessidade de manter um estado dentro de um objeto `Calculadora`.

7. Métodos `static` e o método `main`

Um dos exemplos mais conhecidos de método `static` em Java é o próprio método `main`:

```
public static void main(String[] args) {  
  
}
```

Ele é declarado como:

```
public static
```

porque a JVM precisa conseguir executar o método `main` sem precisar criar previamente um objeto da classe.

Por exemplo:

```
public class Programa {  
  
    public static void main(String[] args) {  
  
        System.out.println("Programa iniciado!");  
    }  
}
```

A JVM consegue executar:

```
Programa.main(args);
```

sem que seja necessário criar:

```
new Programa();
```

8. Métodos `static` não acessam diretamente atributos de instância

Um ponto importante é que um método `static` pertence à classe, enquanto um atributo de instância pertence a um objeto específico.

Considere:

```
public class Pessoa {  
  
    private String nome;  
  
    public static void exibirNome() {  
        System.out.println(nome);  
    }  
}
```

Esse código gera um erro de compilação.

Por quê?

Porque o método `exibirNome()` é `static`, mas `nome` é um atributo de instância.

A classe poderia ter vários objetos:

```
Pessoa  
├─ objeto 1 → nome = "João"  
├─ objeto 2 → nome = "Maria"  
└─ objeto 3 → nome = "Carlos"
```

Qual `nome` deveria ser utilizado pelo método estático?

Não existe um objeto específico associado ao método.

9. Um método `static` pode acessar outro método `static`

Métodos estáticos podem chamar diretamente outros métodos estáticos da mesma classe.

Exemplo:

```
public class Calculadora {  
  
    public static int quadrado(int numero) {  
        return multiplicar(numero, numero);  
    }  
  
    public static int multiplicar(int a, int b) {  
        return a * b;  
    }  
}
```

Utilização:

```
int resultado = Calculadora.quadrado(5);  
  
System.out.println(resultado);
```

Resultado:

25

Como os dois métodos pertencem à classe, a chamada pode ser feita diretamente.

10. Um método `static` pode receber objetos como parâmetros

O fato de um método ser `static` não significa que ele não possa trabalhar com objetos.

Por exemplo:

```
public class Pessoa {  
  
    private String nome;  
  
    public Pessoa(String nome) {  
        this.nome = nome;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
}
```

```
public static void apresentar(Pessoa pessoa) {  
    System.out.println("Olá, " + pessoa.getNome());  
}  
}
```

Podemos utilizar:

```
Pessoa pessoa = new Pessoa("João");  
  
Pessoa.apresentar(pessoa);
```

Resultado:

```
Olá, João
```

Nesse caso, o método é estático, mas recebe explicitamente o objeto que deverá utilizar.

11. Exemplo prático: contador de objetos

Um uso bastante interessante de `static` é criar um contador de objetos.

```
public class Produto {  
  
    private String nome;  
  
    private static int quantidadeProdutos = 0;  
  
    public Produto(String nome) {  
        this.nome = nome;  
        quantidadeProdutos++;  
    }  
  
    public static int getQuantidadeProdutos() {  
        return quantidadeProdutos;  
    }  
}
```

Agora podemos criar produtos:

```
public class Main {  
  
    public static void main(String[] args) {
```

```

        Produto p1 = new Produto("Notebook");
        Produto p2 = new Produto("Mouse");
        Produto p3 = new Produto("Teclado");

        System.out.println(
            Produto.getQuantidadeProdutos()
        );
    }
}

```

Resultado:

3

Observe que `quantidadeProdutos` não pertence a um produto específico. Ela representa uma informação relacionada à **classe Produto como um todo**.

12. Exemplo prático: geração de identificadores

Outra aplicação é gerar identificadores automaticamente.

```

public class Cliente {

    private static int proximoId = 1;

    private int id;
    private String nome;

    public Cliente(String nome) {
        this.id = proximoId++;
        this.nome = nome;
    }

    public int getId() {
        return id;
    }

    public String getNome() {
        return nome;
    }
}

```

Podemos criar vários clientes:

```
Cliente c1 = new Cliente("João");
Cliente c2 = new Cliente("Maria");
Cliente c3 = new Cliente("Carlos");

System.out.println(c1.getId());
System.out.println(c2.getId());
System.out.println(c3.getId());
```

Resultado:

```
1
2
3
```

O atributo:

```
private static int proximoId = 1;
```

é compartilhado entre todos os objetos.

Cada novo cliente utiliza o valor atual e incrementa o contador.

13. Classes utilitárias

Métodos `static` são muito utilizados em classes que fornecem funcionalidades genéricas.

A própria API do Java possui diversos exemplos.

A classe `Math`, por exemplo, possui métodos estáticos:

```
Math.abs(-10);
Math.sqrt(25);
Math.pow(2, 3);
Math.max(10, 20);
```

Por exemplo:

```
double raiz = Math.sqrt(81);

System.out.println(raiz);
```

Resultado:

```
9.0
```

Não precisamos criar:

```
Math math = new Math();
```

As operações são disponibilizadas diretamente pela classe.

14. `static` não significa "global"

É importante não confundir `static` com uma variável global.

Java não possui variáveis globais no mesmo sentido encontrado em algumas outras linguagens.

Um atributo `static` continua pertencendo a uma classe e respeita os mecanismos de encapsulamento e controle de acesso.

Por exemplo:

```
public class Configuracao {  
  
    private static String ambiente = "PRODUCAO";  
  
    public static String getAmbiente() {  
        return ambiente;  
    }  
}
```

O atributo é privado:

```
private static String ambiente;
```

e pode ser acessado através de um método:

```
System.out.println(  
    Configuracao.getAmbiente()  
);
```

15. Diferença entre membro de instância e membro `static`

Considere a classe:

```
public class Conta {  
  
    private String numero;  
  
    private double saldo;  
  
    private static int quantidadeContas;  
  
    public Conta(String numero) {  
        this.numero = numero;  
        quantidadeContas++;  
    }  
}
```

Podemos separar os atributos em dois grupos.

Atributos de instância

```
private String numero;  
private double saldo;
```

Cada objeto possui seus próprios valores:

```
Conta 1  
numero = "001"  
saldo = 1000  
  
Conta 2  
numero = "002"  
saldo = 500
```

Atributo `static`

```
private static int quantidadeContas;
```

Existe apenas uma variável compartilhada:

```
Conta
└─ quantidadeContas = 2
```

16. Cuidados com `static`

Embora `static` seja bastante útil, seu uso deve ser feito com critério.

O uso excessivo de atributos estáticos pode criar um estado compartilhado difícil de controlar.

Por exemplo:

```
public class Sistema {
    public static double saldo;
}
```

Qualquer parte da aplicação poderia alterar esse valor, dependendo da visibilidade:

```
Sistema.saldo = 1000;
```

Isso pode dificultar a manutenção e o teste do sistema.

Em aplicações orientadas a objetos, é geralmente preferível utilizar atributos de instância quando o dado representa o **estado de um objeto**.

17. Regra prática

Uma boa regra para decidir entre um membro de instância e um membro `static` é perguntar:

"Esse dado ou comportamento pertence a cada objeto ou à classe como um todo?"

Se pertence a cada objeto:

```
private String nome;
private double salario;
private LocalDate dataNascimento;
```

provavelmente deve ser um atributo de instância.

Se pertence à classe:

```
private static int quantidade;  
public static int somar(int a, int b);
```

pode ser apropriado utilizar `static`.

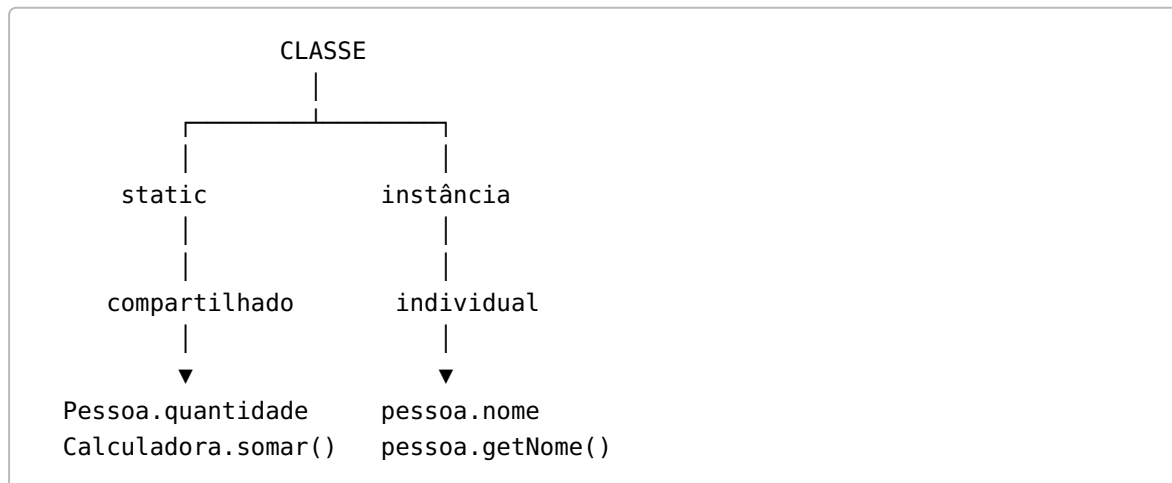
Conclusão

O modificador `static` permite criar atributos e métodos que pertencem à **classe**, e não a uma instância específica.

Os principais conceitos são:

- `static` indica que o membro pertence à classe;
- atributos `static` são compartilhados entre os objetos;
- métodos `static` podem ser chamados sem criar objetos;
- métodos `static` não acessam diretamente atributos de instância;
- `static final` é muito utilizado para criação de constantes;
- métodos `static` são comuns em classes utilitárias;
- o método `main` é um exemplo importante de método `static`;
- `static` deve ser utilizado com critério para evitar estado compartilhado desnecessário.

Em resumo:



Compreender a diferença entre membros `static` e membros de instância é fundamental para utilizar corretamente os recursos da programação orientada a objetos em Java.