

Métodos: Declaração, Parâmetros e Sobrecarga

Como definir e reutilizar comportamentos em Java

1. Introdução

Um **método** é um bloco de código nomeado que executa uma tarefa específica e pode ser chamado (invocado) sempre que essa tarefa for necessária. Métodos são a principal forma de organizar o comportamento de uma classe em Java, evitando a repetição de código e tornando os programas mais legíveis e fáceis de manter. Este texto explica como declarar métodos, como utilizar parâmetros e como aplicar a sobrecarga de métodos, sempre com exemplos práticos em Java.

2. Declaração de Métodos

A declaração de um método em Java segue uma estrutura bem definida, composta por modificador de acesso, tipo de retorno, nome, lista de parâmetros (entre parênteses) e o corpo do método (entre chaves).

```
modificador tipoDeRetorno nomeDoMetodo(parametros) {  
    // corpo do método  
    return valor; // obrigatório se o tipo de retorno não for void  
}
```

Por exemplo, um método que soma dois números inteiros e devolve o resultado:

```
public class Calculadora {  
    public int somar(int a, int b) {  
        int resultado = a + b;  
        return resultado;  
    }  
}  
  
// Uso:  
Calculadora calc = new Calculadora();  
int total = calc.somar(5, 3);  
System.out.println(total); // 8
```

2.1. Métodos void

Quando um método não precisa devolver nenhum valor, seu tipo de retorno é declarado como **void**. Nesse caso, a instrução **return** é opcional e, se usada, não deve vir acompanhada de nenhum valor.

```
public class Impressora {
    public void imprimirMensagem(String mensagem) {
        System.out.println("Mensagem: " + mensagem);
        // não há necessidade de "return", pois o tipo é void
    }
}
```

2.2. Métodos static vs métodos de instância

Métodos declarados com o modificador `static` pertencem à classe e podem ser chamados sem a criação de um objeto. Já os métodos de instância (sem `static`) só podem ser chamados a partir de um objeto criado com `new`, pois geralmente operam sobre o estado desse objeto específico.

```
public class MathUtils {
    // Método estático: não depende de um objeto
    public static int dobro(int numero) {
        return numero * 2;
    }
}

public class Contador {
    private int valor;

    // Método de instância: depende do estado do objeto
    public void incrementar() {
        valor++;
    }
}

// Uso:
int resultado = MathUtils.dobro(10); // chamado direto pela classe

Contador c = new Contador();
c.incrementar(); // precisa de um objeto
```

3. Parâmetros

Os **parâmetros** permitem que um método receba valores externos para trabalhar com eles em vez de sempre usar dados fixos. Cada parâmetro é declarado com um tipo e um nome, e os valores passados na chamada do método são chamados de **argumentos**.

```
public class Saudacao {
    public void cumprimentar(String nome, int idade) { // parâmetros
        System.out.println("Olá, " + nome + "! Você tem " + idade + " anos.");
    }
}

Saudacao s = new Saudacao();
s.cumprimentar("Ana", 25); // "Ana" e 25 são os argumentos
```

3.1. Parâmetros do tipo Varargs

Java permite declarar um parâmetro do tipo **varargs** ("argumentos de tamanho variável"), usando três pontos (`...`) após o tipo. Isso permite chamar o método passando qualquer quantidade de

argumentos daquele tipo, inclusive nenhum, e dentro do método eles são tratados como um array.

```
public class MathUtils {
    public static int somarTodos(int... numeros) {
        int soma = 0;
        for (int n : numeros) {
            soma += n;
        }
        return soma;
    }
}

System.out.println(MathUtils.somarTodos(1, 2, 3));        // 6
System.out.println(MathUtils.somarTodos(10, 20));        // 30
System.out.println(MathUtils.somarTodos());              // 0
```

3.2. Passagem de Parâmetros: Valor vs Referência

Java sempre passa argumentos **por valor**. Para tipos primitivos, o método recebe uma cópia do valor; para tipos por referência (objetos e arrays), o método recebe uma cópia da referência, que ainda aponta para o mesmo objeto na memória.

```
public class Main {
    static void dobrar(int numero) {
        numero = numero * 2; // altera apenas a cópia local
    }

    static void adicionarItem(java.util.List<String> lista) {
        lista.add("novo item"); // altera o objeto compartilhado
    }

    public static void main(String[] args) {
        int x = 5;
        dobrar(x);
        System.out.println(x); // 5 (não mudou)

        java.util.List<String> itens = new java.util.ArrayList<>();
        adicionarItem(itens);
        System.out.println(itens); // [novo item] (objeto foi alterado)
    }
}
```

4. Sobrecarga de Métodos (Overloading)

A **sobrecarga de métodos** permite que uma classe tenha vários métodos com o **mesmo nome**, desde que suas **listas de parâmetros** sejam diferentes — seja em quantidade, tipo ou ordem dos parâmetros. O tipo de retorno, isoladamente, **não** é suficiente para diferenciar dois métodos sobrecarregados.

```

public class Calculadora {

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

    public double somar(double a, double b) {
        return a + b;
    }

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

    public String somar(String a, String b) {
        return a + b; // concatenação, não soma numérica
    }
}

Calculadora calc = new Calculadora();
System.out.println(calc.somar(2, 3));           // 5    (int, int)
System.out.println(calc.somar(2.5, 3.1));       // 5.6  (double, double)
System.out.println(calc.somar(1, 2, 3));        // 6    (int, int, int)
System.out.println(calc.somar("Java", "P00"));  // JavaP00

```

O compilador Java decide, em tempo de compilação, qual versão do método `somar` deve ser chamada, com base no número e no tipo dos argumentos informados. Esse mecanismo é chamado de **polimorfismo estático** (ou *resolução em tempo de compilação*), em contraste com o polimorfismo dinâmico obtido por meio da sobrescrita (*override*) de métodos em herança.

4.1. O que NÃO caracteriza sobrecarga

Alterar apenas o tipo de retorno, mantendo os mesmos parâmetros, não é considerado sobrecarga e resulta em erro de compilação, pois o compilador não teria como saber qual método chamar apenas pela chamada.

```

public class Exemplo {
    public int calcular(int a, int b) {
        return a + b;
    }

    // ERRO de compilação: mesmo nome e mesmos parâmetros,
    // apenas o tipo de retorno é diferente
    // public double calcular(int a, int b) {
    //     return a + b;
    // }
}

```

5. Exemplo Integrado

O exemplo a seguir reúne declaração de métodos, parâmetros variados e sobrecarga em uma única classe utilitária para formatação de mensagens.

```

public class Formatador {

    public static String formatar(String texto) {
        return "[" + texto + "]";
    }

    public static String formatar(String texto, String prefixo) {
        return prefixo + ": [" + texto + "]";
    }

    public static String formatar(String... textos) {
        StringBuilder sb = new StringBuilder();
        for (String t : textos) {
            sb.append("[").append(t).append("] ");
        }
        return sb.toString().trim();
    }

    public static void main(String[] args) {
        System.out.println(formatar("Aviso"));
        System.out.println(formatar("Erro", "Sistema"));
        System.out.println(formatar("A", "B", "C"));
    }
}
// Saída:
// [Aviso]
// Sistema: [Erro]
// [A] [B] [C]

```

6. Conclusão

Métodos bem declarados, com parâmetros claros e bem definidos, são a base para um código Java organizado e reutilizável. A sobrecarga de métodos amplia essa flexibilidade, permitindo que uma mesma operação lógica (como "somar" ou "formatar") seja aplicada a diferentes tipos e quantidades de dados sem a necessidade de nomes distintos para cada variação, tornando a interface da classe mais intuitiva para quem a utiliza.