

Inicializando Objetos com Construtores

Como criar e configurar objetos corretamente em Java

1. Introdução

Um **construtor** é um bloco de código especial, executado automaticamente sempre que um objeto é criado com a palavra-chave `new`. Sua principal função é inicializar o estado do objeto, ou seja, atribuir valores iniciais aos seus atributos, garantindo que o objeto comece a existir em um estado válido e consistente.

Um construtor tem o mesmo nome da classe, não possui tipo de retorno (nem mesmo `void`) e pode receber parâmetros, assim como um método comum.

2. Construtor Padrão (Default)

Quando uma classe não declara nenhum construtor, o compilador Java gera automaticamente um **construtor padrão**, sem parâmetros, que inicializa os atributos com seus valores padrão (0, `false`, `null`, conforme o tipo).

```
public class Produto {  
    private String nome;  
    private double preco;  
    // Nenhum construtor declarado  
}  
  
// O compilador gera, implicitamente, algo equivalente a:  
// public Produto() { }  
  
Produto p = new Produto();  
System.out.println(p.getNome()); // null  
System.out.println(p.getPreco()); // 0.0
```

Assim que a classe declara **qualquer** construtor próprio, o construtor padrão deixa de ser gerado automaticamente pelo compilador.

3. Construtor com Parâmetros

Um construtor pode receber parâmetros para inicializar os atributos do objeto com valores específicos no momento da criação, evitando a necessidade de chamar vários métodos `set` logo em seguida.

```

public class Produto {
    private String nome;
    private double preco;

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

    public String getNome() { return nome; }
    public double getPreco() { return preco; }
}

public class Main {
    public static void main(String[] args) {
        Produto p = new Produto("Caderno", 15.90);
        System.out.println(p.getNome() + " - R$ " + p.getPreco());
    }
}

```

A palavra-chave `this` é usada para diferenciar o atributo da classe (`this.nome`) do parâmetro do construtor que possui o mesmo nome (`nome`).

4. Sobrecarga de Construtores

Assim como métodos, os construtores podem ser **sobrecarregados**: uma classe pode ter vários construtores, desde que cada um tenha uma lista de parâmetros diferente (em quantidade e/ou tipo). Isso permite criar objetos de diferentes formas, conforme a informação disponível no momento da criação.

```

public class Produto {
    private String nome;
    private double preco;
    private int quantidade;

    public Produto(String nome) {
        this.nome = nome;
        this.preco = 0.0;
        this.quantidade = 0;
    }

    public Produto(String nome, double preco) {
        this.nome = nome;
        this.preco = preco;
        this.quantidade = 0;
    }

    public Produto(String nome, double preco, int quantidade) {
        this.nome = nome;
        this.preco = preco;
        this.quantidade = quantidade;
    }
}

// Uso:
Produto p1 = new Produto("Caneta");
Produto p2 = new Produto("Caderno", 15.90);
Produto p3 = new Produto("Mochila", 89.90, 5);

```

5. Encadeamento de Construtores com this()

Para evitar repetir código entre construtores sobrecarregados, um construtor pode chamar outro construtor da mesma classe usando `this(...)` como a **primeira instrução** do seu corpo. Isso é chamado de **encadeamento de construtores** e ajuda a centralizar a lógica de inicialização em um único lugar.

```
public class Produto {
    private String nome;
    private double preco;
    private int quantidade;

    public Produto(String nome, double preco, int quantidade) {
        this.nome = nome;
        this.preco = preco;
        this.quantidade = quantidade;
    }

    public Produto(String nome, double preco) {
        this(nome, preco, 0); // reaproveita o construtor mais completo
    }

    public Produto(String nome) {
        this(nome, 0.0); // encadeia para o construtor anterior
    }
}
```

Nesse exemplo, os construtores mais simples delegam a inicialização para o construtor mais completo, evitando duplicar as atribuições `this.nome = nome`, `this.preco = preco`, etc. em cada um deles.

6. A Palavra-chave `super()` em Herança

Quando uma classe estende outra, o construtor da subclasse pode (e, em muitos casos, deve) chamar explicitamente um construtor da superclasse usando `super(...)`, também como a primeira instrução do construtor. Se nenhuma chamada a `super(...)` for feita explicitamente, o Java insere automaticamente uma chamada a `super()` sem argumentos.

```

public class Pessoa {
    protected String nome;

    public Pessoa(String nome) {
        this.nome = nome;
        System.out.println("Construtor de Pessoa executado.");
    }
}

public class Aluno extends Pessoa {
    private String matricula;

    public Aluno(String nome, String matricula) {
        super(nome); // chama o construtor de Pessoa
        this.matricula = matricula;
        System.out.println("Construtor de Aluno executado.");
    }
}

// Saída ao criar: new Aluno("Ana", "2024001");
// Construtor de Pessoa executado.
// Construtor de Aluno executado.

```

*Note que o construtor da superclasse é sempre executado **antes** do construtor da subclasse, garantindo que a parte herdada do objeto já esteja inicializada quando o restante da inicialização começar.*

7. Construtor de Cópia

Embora Java não tenha um construtor de cópia automático (como em C++), é comum criar um construtor que recebe um objeto da mesma classe como parâmetro, copiando seus valores para o novo objeto. Isso é útil para criar um novo objeto independente, com os mesmos dados de outro já existente.

```

public class Produto {
    private String nome;
    private double preco;

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

    // Construtor de cópia
    public Produto(Produto outro) {
        this.nome = outro.nome;
        this.preco = outro.preco;
    }
}

Produto original = new Produto("Caderno", 15.90);
Produto copia = new Produto(original); // novo objeto, com os mesmos dados

```

8. Atributos final e Construtores

Atributos declarados como `final` precisam receber seu valor uma única vez — no momento da declaração ou dentro de um construtor — e nunca mais podem ser alterados depois. Isso os torna

ideais para representar dados que não devem mudar após a criação do objeto, como o CPF de uma pessoa.

```
public class Usuario {
    private final String cpf; // deve ser inicializado no construtor

    public Usuario(String cpf) {
        this.cpf = cpf; // única atribuição permitida
    }

    public String getCpf() {
        return cpf;
    }
}

// Usuario u = new Usuario("123.456.789-00");
// u.cpf = "outro"; // ERRO de compilação: cpf é final
```

9. Boas Práticas

- Utilize o construtor para garantir que o objeto nasça em um **estado válido**, evitando objetos incompletos ou inconsistentes.
- Prefira o **encadeamento com this(...)** a repetir código de inicialização em vários construtores.
- Sempre que uma subclasse depender de dados da superclasse, use `super(...)` explicitamente para deixar clara a inicialização herdada.
- Valide os parâmetros recebidos pelo construtor sempre que fizer sentido, lançando exceções para valores inválidos (por exemplo, um preço negativo).

```
public class ContaBancaria {
    private double saldoInicial;

    public ContaBancaria(double saldoInicial) {
        if (saldoInicial < 0) {
            throw new IllegalArgumentException("Saldo inicial não pode ser negativo.");
        }
        this.saldoInicial = saldoInicial;
    }
}
```

10. Conclusão

Os construtores são a porta de entrada para a criação de objetos consistentes em Java. Seja usando o construtor padrão, construtores sobrecarregados, encadeamento com `this(...)`, herança com `super(...)` ou um construtor de cópia, o objetivo é sempre o mesmo: garantir que, assim que o objeto é criado, ele já esteja pronto para uso, com todos os seus atributos corretamente inicializados.