Sistema Brownies daora: Desvendando a POO em Java

Uma demonstração prática dos pilares da Programação Orientada a Objetos, aplicada a um delicioso sistema de gerenciamento de brownies.

1

Herança e Polimorfismo

Como classes se relacionam e objetos assumem múltiplas formas.

2

Classes Abstratas vs. Interfaces

Entendendo as diferenças e aplicações de cada conceito.

3

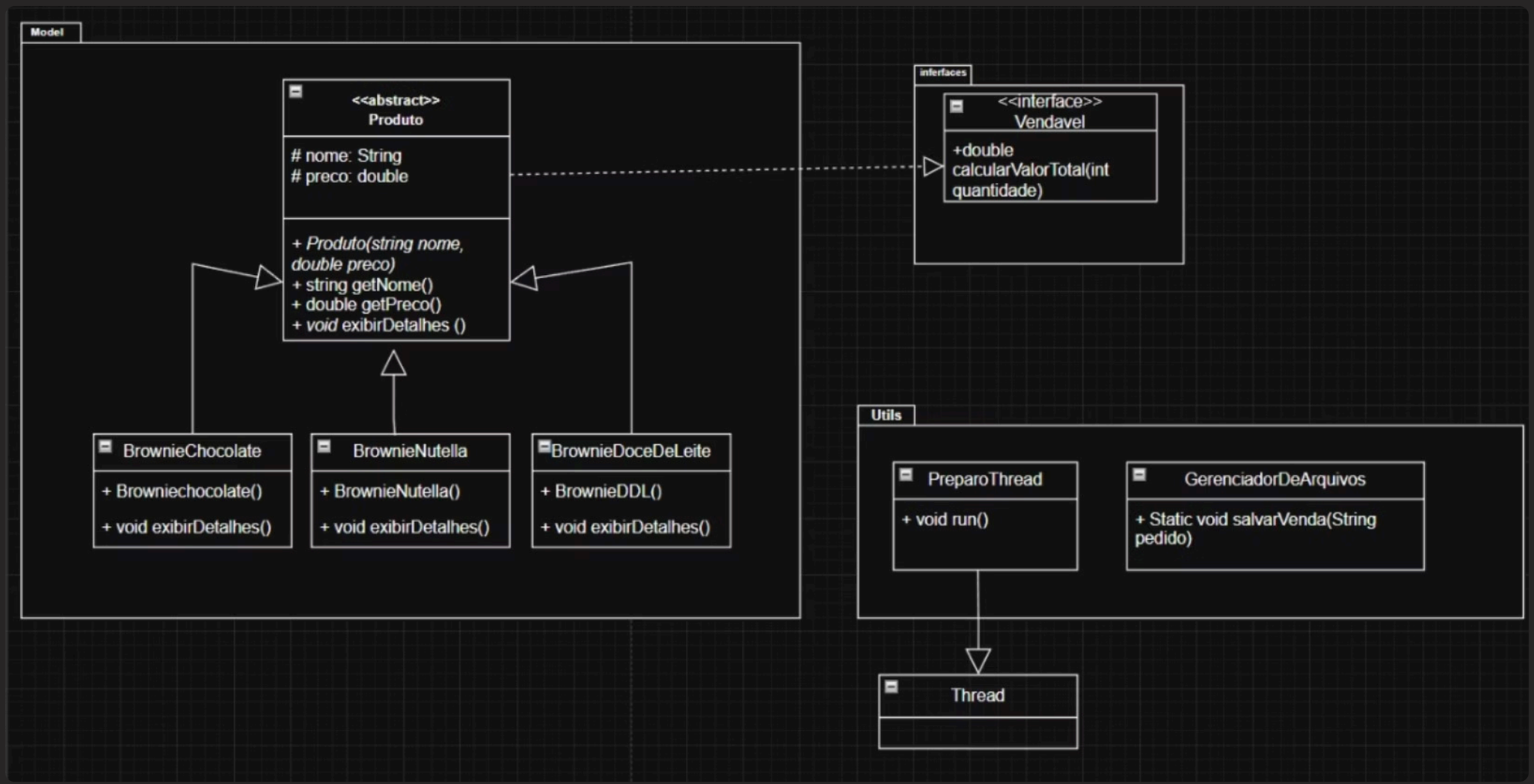
Tratamento de Erros e Threads

Garantindo a robustez e responsividade da aplicação.

4

Persistência em Arquivos

Armazenando e recuperando dados de forma eficaz.



Arquitetura do Sistema: Uma Visão Geral

Nosso sistema de brownies é organizado em pacotes modulares, facilitando o desenvolvimento e a manutenção. Cada componente tem uma função específica, demonstrando boas práticas de design de software.

Pacotes Principais: Interface/ (Define contratos para funcionalidades), Model/ (Contém as classes de dados e lógica de negócio), Utils/ (Utilitários para operações diversas), Main/ (Ponto de entrada da aplicação).

Interfaces vs. Classes Abstratas:

Na programação orientada a objetos, entender a diferença entre interfaces e classes abstratas é crucial. Cada uma serve a um propósito distinto, e a escolha correta impacta diretamente a flexibilidade e manutenibilidade do seu código.

Interface Vendavel:

```java
public interface Vendavel {

    public double calcularValorTotal(int quantidade);
}
```

A interface Vendavel define um contrato com métodos puramente abstratos e públicos, sem implementação ou membros de dados. Ela assegura que classes que a implementem sigam um padrão específico, focando apenas no "o quê" fazer.

Classe Abstrata Produto:

```java
public abstract class Produto implements Vendavel {
    protected String nome;
    protected double precoUnitario;

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

    // Getters
    public String getNome() {
        return nome;
    }
    public double getPrecoUnitario() {
        return precoUnitario;
    }

    // Implementação da interface Vendavel.
    @Override
    public double calcularValorTotal(int quantidade) {
        return this.precoUnitario * quantidade;
    }

    //Metodo Abstrato: Ele OBRIGA cada brownie específico a criar sua própria mensagem
    public abstract void exibirDetalhes();
}
```

A classe abstrata Produto oferece uma base comum com implementação parcial e métodos abstratos. Ela permite compartilhar código e atributos (como nome e preço), enquanto obriga subclasses a implementar métodos específicos, como a exibição de detalhes, que podem variar.

Herança e Especialização: Brownies Sob Medida

A herança nos permite criar uma hierarquia de classes, onde classes filhas (os tipos de brownie) herdam características e comportamentos da classe pai (Produto), mas também adicionam suas próprias particularidades.

```java
public class BrownieNutella extends Produto{
    public BrownieNutella() {
        // 'super' Chama o construtor da classe pai (Produto)
        super(nome: "Brownie de Nutella", preco: 10.00);
    }

    @Override
    public void exibirDetalhes() {
        System.out.println(x: "--- DETALHES DO PRODUTO ---");
        System.out.println("Sabor: " + this.nome);
        System.out.println("Preço: R$ " + this.precoUnitario);
        System.out.println(x: "Descrição: Delicioso brownie recheado com muuita Nutella.");
    }
}
```

Brownie chocolate

Clássico e intenso, herda de Produto e implementa Vendavel.

Brownie de doce de Leite

Uma variação com sabor de caramelo, estendendo a funcionalidade do brownie base.

Brownie Nutella

Elegante e colorido, adicionando um toque especial à receita original.

Cada brownie mantém os atributos básicos de um produto, mas pode ter implementações específicas para métodos como exibirDetalhes(), conforme a necessidade. Assim, conseguimos reutilizar código e manter uma estrutura organizada.

Polimorfismo na Prática: Flexibilidade em Tempo de Execução

O polimorfismo, que significa "muitas formas", nos permite tratar objetos de diferentes classes de maneira uniforme, desde que eles compartilhem uma mesma base (interface ou classe abstrata). No nosso sistema, isso é fundamental para gerenciar os diferentes tipos de brownies.

```
Produto produtoEscolhido = null;

switch (opcao) {
    case 1:
        produtoEscolhido = new BrownieNutella();
        break;
    case 2:
        produtoEscolhido = new BrownieDoceDeLeite();
        break;
    case 3:
        produtoEscolhido = new BrownieChocolate();
        break;
    default:
        System.out.println(x: "Opção inválida!");
        continue; // Volta para o início do loop
}

// Se chegou aqui, temos um produto válido
System.out.print(s: "Digite a quantidade: ");
int qtd = scanner.nextInt();

// 1. Exibir detalhes (Polimorfismo em ação)
produtoEscolhido.exibirDetalhes();

// 2. Calcular total (Uso da Interface)
double valorTotal = produtoEscolhido.calcularValorTotal(qtd);
```

Ao declarar uma variável do tipo Vendavel ou Produto, podemos atribuir a ela qualquer um dos nossos brownies especializados (BrownieChocolate, BrownieDoceDeLeite, BrownieRedVelvet). Quando chamamos um método como exibirDetalhes(), o comportamento específico de cada brownie é invocado em tempo de execução.

Tratamento de Erros: Robustez com Try-Catch

Em qualquer aplicação, erros são inevitáveis. O tratamento de exceções é fundamental para que o sistema não falhe inesperadamente e possa se recuperar gracefully, ou pelo menos informar o usuário de forma amigável.

```
try {
    opcao = scanner.nextInt();
    // Processamento normal
} catch (InputMismatchException e) {
    System.out.println("ERRO: Digite apenas números!");
    scanner.nextLine(); // Limpa buffer
} catch (Exception e) {
    System.out.println("Erro inesperado: " + e.getMessage());
}
```

Utilizamos blocos try-catch para capturar e gerenciar possíveis erros, como a leitura ou escrita de arquivos. Isso garante que a aplicação continue funcionando mesmo diante de imprevistos, oferecendo uma experiência mais estável ao usuário.

Lembrete: esse código acima é uma versão simplificada que contem exclusivamente o tratamento de erro!!

Persistência em Arquivo: Salvando Dados com GerenciadorDeArquivos

Para que os dados dos nossos brownies não se percam ao desligar a aplicação, implementamos um sistema simples de persistência em arquivos. A classe GerenciadorDeArquivos é responsável por escrever e ler informações em arquivos de texto.

- Salva a lista de brownies vendidos.
- Recupera os registros ao iniciar a aplicação.
- Utiliza BufferedWriter e BufferedReader para operações eficientes.

```
public static void salvarVenda(String detalheVenda) {

    // Formata a data e hora atual para o registro ficar bonito no arquivo
    DateTimeFormatter dtf = DateTimeFormatter.ofPattern(pattern: "dd/MM/yyyy HH:mm:ss");
    String dataHora = dtf.format(LocalDateTime.now());

    // Monta a linha final que vai para o arquivo
    String linhaRegistro = "[" + dataHora + "] " + detalheVenda;

    try (FileWriter fw = new FileWriter(fileName: "vendas.txt", append: true);
         PrintWriter pw = new PrintWriter(fw)) {

        pw.println(linhaRegistro);
        System.out.println(x: ">> Venda salva no arquivo 'vendas.txt' com sucesso.");

    } catch (IOException e) {
        // Se der erro (ex: disco cheio), o programa não trava, apenas avisa.
        System.err.println(x: "ERRO CRÍTICO: Não foi possível salvar a venda no arquivo.");
        System.err.println("Detalhe do erro: " + e.getMessage());
    }
}
```

Programação Concorrente

Para simular um processo de preparo de brownie que não bloqueie a interface do usuário, empregamos a programação concorrente com Threads. A classe PreparoThread executa o processo em segundo plano.

```java
public class PreparoThread extends Thread {

    // O metodo run() é onde a mágica da Thread acontece.
    // Tudo que está aqui dentro roda separadamente do fluxo principal se necessário.
    @Override
    public void run() {
        try {
            System.out.println(x: "\n[SISTEMA DE COZINHA] Iniciando preparo do Brownie...");
            System.out.print(s: "Processando: ");

            // Loop para simular uma barra de progresso
            for (int i = 0; i < 5; i++) {
                // Pausa a execução desta thread por 1 segundo (1000 ms)
                Thread.sleep(millis: 1000);
                System.out.print(s: " ▰"); // Caractere visual de barra
            }

            System.out.println(x: "\n[SISTEMA DE COZINHA] Pedido finalizado e embalado!\n");

        } catch (InterruptedException e) {
            // Tratamento obrigatório para Threads (caso o processo seja interrompido bruscamente)
            // Critério atendido: Tratamento de erros

            System.out.println("Erro: O preparo foi interrompido! " + e.getMessage());
        }
    }
}
```

Na classe Main, instanciamos e iniciamos a thread

```java
// 3. Iniciar a Thread de preparo (Critério 8 e 16)
PreparoThread preparo = new PreparoThread();
preparo.start();

/* * O join() faz o programa principal "esperar" a Thread terminar
 * para só depois salvar o arquivo. Isso sincroniza as coisas.
 */
try {
    preparo.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Fluxo de Execução: A Jornada do Brownie

Vamos visualizar como todos esses conceitos se unem no nosso sistema, desde a escolha do cliente até a finalização do pedido e a persistência dos dados.

Menu Interativo

O usuário seleciona o brownie desejado através de um menu amigável.

Criação Polimórfica

O brownie é instanciado usando polimorfismo, tratado como um Produto ou Vendavel.

Cálculo do Total

O preço final é calculado usando o método da interface Vendavel.

Exibição Detalhada

Os detalhes do brownie são exibidos através do método abstrato implementado.

Preparo Assíncrono

Uma Thread separada simula o tempo de preparo do brownie.

Salvamento de Dados

O registro da venda é salvo em um arquivo de texto pelo GerenciadorDeArquivos.

Retorno ao Menu

A aplicação retorna ao menu principal, pronta para um novo pedido.

Conclusão: Conceitos de POO Demonstrados na Prática

O sistema "Brownies daora" ilustra de forma concisa e saborosa como os principais pilares da Programação Orientada a Objetos podem ser aplicados em um projeto real, resultando em um código mais organizado, flexível e robusto.

✅ Abstração:

Utilizamos Interfaces e Classes Abstratas para definir contratos e modelos.

✅ Encapsulamento:

Atributos protected garantem o acesso controlado aos dados.

✅ Herança:

Brownies especializados herdam de Produto, reutilizando código e lógica.

✅ Polimorfismo:

Objetos de tipos diferentes são tratados de forma uniforme, adicionando flexibilidade.

✅ Tratamento de Erros:

Blocos try-catch garantem a robustez da aplicação.

✅ Persistência:

Gerenciamento de dados em arquivos de texto para armazenamento.

✅ Concorrência:

Threads permitem processos paralelos e uma UI responsiva.

Perguntas? 🍫