UNIVERSIDADE PITÁGORAS UNOPAR

Engenharia de Software
Sistemas Operacionais

AUTOR: ALLAS MAYCON DO VALLE
RA: 3649457105

Programação Orientada a Objetos II

Programação em Java usando threads

Cidade de Serrana, SP
2025

AUTOR: ALLAS MAYCON DO VALLE
RA: 3649457105

Programação Orientada a Objetos II

Programação em Java usando threads

O objetivo aqui foi fazer o nosso programa Java rodar duas tarefas diferentes ao mesmo tempo, ou seja, em paralelo. Para isso, seguimos criando duas classes, e cada uma sabia fazer uma única tarefa. A primeira classe foi a Contadora, que simplesmente conta de 1 a 10 e espera 1 segundo, após cada número. A segunda foi a Mensageira, que imprime as frases que colocamos dentro, esperando um pouco mais de 2 segunda entre elas.
Orientador:
Tutor à Distância: Frederico Aparecido Faedo Pinto.
Prof. Renan Cleverson Laureano Flor da Rosa.

Cidade de Serrana, SP
2025

SUMÁRIO

1 INTRODUÇÃO 3
2 DESENVOLVIMENTO 4
2.1 METODOLOGIA DE IMPLEMENTAÇÃO 4
2.2 CÓDIGO-FONTE COMPLETO 4
2.3 DETALHAMENTO DA CONFORMIDADE COM OS REQUISITOS 5
3 CONCLUSÃO 6
4 REFERÊNCIAS 7
5 APÊNDICE: REPOSITÓRIO DO CÓDIGO FONTE 7

1 INTRODUÇÃO

Este projeto prático marca o nosso avanço da programação de interfaces visuais (como no Projeto 1) para a área de **programação concorrente** em Java. O tema central é o uso de **Threads**.

O principal objetivo deste projeto foi entender como fazer um programa realizar várias coisas ao mesmo tempo. Normalmente, um programa Java roda de cima para baixo em uma única linha, chamada *thread principal*. Ao usar *múltiplas threads*, podemos fazer com que diferentes partes do nosso código rodem em paralelo, o que é muito útil para que o programa não "trave" enquanto espera por alguma tarefa lenta.

Para atender aos requisitos, decidimos criar um cenário simples, mas que demonstra claramente a concorrência:

- Thread A (Contadora): Vai mostrar números de 1 até 10. Para dar tempo de vermos o que está acontecendo, ela vai esperar **1 segundo** entre um número e outro.

- Thread B (Mensagens): Vai mostrar mensagens que nós mesmos criamos, como "Processando..." ou "Aguarde". Ela vai esperar **2 segundos** entre cada mensagem.

A metodologia de implementação seguiu a melhor prática para iniciantes, utilizando a interface **Runnable**. Isso permite que o código da tarefa fique separado da classe Thread em si.

O procedimento envolveu: criar duas classes Runnable (uma para cada tarefa), usar o método run() para colocar o código de contagem/mensagens, e, principalmente, incluir o comando **Thread.sleep()** para criar os atrasos pedidos (1 e 2 segundos). Finalmente, usamos a classe Thread para iniciar as duas tarefas ao mesmo tempo (start()), comprovando a execução simultânea.

O resultado esperado é ver a contagem e as mensagens aparecendo no console de forma misturada, provando que o Java está executando as duas tarefas em paralelo, sem precisar esperar que uma acabe para a outra começar.

2 DESENVOLVIMENTO

A implementação deste projeto foi dividida em três classes distintas para garantir que cada tarefa fosse bem separada, como é uma boa prática em Java. Utilizamos a interface Runnable nas classes de tarefa, conforme solicitado nos procedimentos, e a classe principal (TesteConcorrencia) para orquestrar a execução.

2.1 Classe 1: TarefaContadora (Implementa Runnable)

Esta classe é responsável por contar de 1 a 10 e introduzir o atraso de 1 segundo (1000 milissegundos).

Java
```
import java.util.concurrent.TimeUnit;

public class TarefaContadora implements Runnable {

    @Override
    public void run() {
        String nomeThread = Thread.currentThread().getName();
        System.out.println(nomeThread + " - INICIADA: Vai contar de 1 a 10.");

        try {
            for (int i = 1; i <= 10; i++) {
                System.out.println(nomeThread + " - CONTAGEM: " + i);

                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.err.println(nomeThread + " foi interrompida de forma inesperada.");
        }

        System.out.println(nomeThread + " - FINALIZADA.");
    }
}
```

2.2 Classe 2: TarefaMensagens (Implementa Runnable)

Esta classe exibe as mensagens personalizadas e utiliza o atraso de 2 segundos (2000 milissegundos).

Java
```
public class TarefaMensagens implements Runnable {
    private String[] mensagens = {
        "Processando dados...",
        "Aguardando resposta do servidor...",
        "Quase finalizando a tarefa!",
        "Checagem de status concluída.",
        "Aguarde o encerramento da contagem..."
    };
```

```
    @Override
    public void run() {
        String nomeThread = Thread.currentThread().getName();
        System.out.println(nomeThread + " - INICIADA: Vai exibir mensagens.");

        try {
            for (String msg : mensagens) {
                System.out.println(nomeThread + " - MENSAGEM: " + msg);

                Thread.sleep(2000);
            }
        } catch (InterruptedException e) {
            System.err.println(nomeThread + " foi interrompida de forma inesperada.");
        }

        System.out.println(nomeThread + " - FINALIZADA.");
    }
}
```

2.3 Classe 3: TesteConcorrencia (Classe Principal)

Esta classe é o ponto de entrada (main) e é responsável por criar os objetos Thread a partir dos objetos Runnable e iniciar a execução simultânea.

Java

```
public class TesteConcorrencia {

    public static void main(String[] args) {

        System.out.println("Programa Principal INICIADO.\n");

        // 1. Cria a tarefa de contagem
        TarefaContadora tarefaContadora = new TarefaContadora();
        // 2. Cria a thread T1 com a tarefa e da um nome claro
        Thread tContadora = new Thread(tarefaContadora, "T1-Contadora");

        // 3. Cria a tarefa de mensagens
        TarefaMensagens tarefaMensagens = new TarefaMensagens();
        // 4. Cria a thread T2 com a tarefa e da um nome claro
        Thread tMensagens = new Thread(tarefaMensagens, "T2-Mensagens");

        // REQUISITO: Iniciar as threads de forma SIMULTÂNEA
        tContadora.start();
        tMensagens.start();

        System.out.println("\nTodas as threads filhas foram iniciadas (Simultaneidade garantida).\n");

        // Fim da execucao do main. O programa so encerra quando T1 e T2 terminarem.
    }
}
```

2.4 Análise do Resultado (Saída Esperada)

Ao executar o código, o console exibirá o *output* das threads T1-Contadora e T2-Mensagens de forma misturada e não sequencial (não determinística). Por exemplo:

- T1: Contagem 1 (depois de 1s)

- T2: Mensagem 1 (depois de 2s)

- T1: Contagem 2 (depois de mais 1s)

- T1: Contagem 3 (depois de mais 1s)

- T2: Mensagem 2 (depois de mais 2s)

Essa intercalação comprova que as tarefas estão sendo executadas de forma **simultânea** e que os métodos Thread.sleep() estão funcionando conforme os requisitos de tempo (1 segundo e 2 segundos).

3 CONCLUSÃO

O projeto foi finalizado com êxito, cumprindo todos os objetivos definidos para a demonstração prática de Threads em Java. O principal aprendizado desta atividade foi entender o conceito de **concorrência**, que é a capacidade de um programa realizar mais de uma tarefa ao mesmo tempo, sem que uma precise esperar a outra terminar.

Nós conseguimos provar que a execução paralela funciona ao criar e iniciar as duas threads:

- A **T1-Contadora**, que executava sua contagem de 1 a 10 com atrasos de 1 segundo.

- A **T2-Mensagens**, que exibia suas mensagens a cada 2 segundos.

Ao rodar o programa, o *output* no console mostrou as mensagens da Thread 1 e da Thread 2 aparecendo intercaladas. Isso prova que o comando **tContadora.start()** e o **tMensagens.start()** realmente funcionaram para iniciar as duas tarefas de forma simultânea. Se tivéssemos rodado o código sequencialmente, a Thread 2 só começaria depois que a Thread 1 terminasse a contagem completa.

A escolha de implementar a interface **Runnable** para criar as tarefas (em vez de estender a classe Thread) foi a forma mais limpa de fazer isso, pois separa o código da tarefa do código do controle da Thread. Isso é uma boa prática que aprendemos a usar.

Outro ponto fundamental foi o uso correto do **Thread.sleep(tempo)**. Este método foi essencial para criarmos os intervalos de 1 e 2 segundos pedidos. Ele serviu para simular que as threads estavam ocupadas com alguma tarefa demorada, garantindo que o efeito da concorrência fosse claramente visível no console.

O projeto estabeleceu uma base sólida em programação multithreading.

Próximas melhorias e aprendizados futuros:

Embora o projeto tenha funcionado bem, ele ainda é simples porque as threads não precisam compartilhar dados. Em um projeto mais avançado, o próximo passo seria:

- Introduzir a **sincronização** (synchronized ou Lock) para evitar que as threads "briguem" por um mesmo recurso.

- Utilizar o método **join()** na thread principal para garantir que o programa só finalize depois que as threads filhas terminarem, de forma controlada.

Em resumo, demonstramos a criação, execução e temporização de múltiplas linhas de execução, provando que agora entendemos como fazer um programa rodar em paralelo.

REFERÊNCIAS

DEITEL, Harvey M.; DEITEL, Paul J. Livro - Java: Como Programar. 10. ed. São Paulo: Pearson Education do Brasil, 2016. (Esta é a referência que você forneceu, essencial para fundamentos de Java e que costuma cobrir threads em capítulos avançados.)

GOETZ, Brian et al. Java Concurrency in Practice. 1. ed. Boston: Addison-Wesley Professional, 2006. (Este livro é a referência definitiva e mais citada mundialmente sobre concorrência e multithreading em Java, perfeito para aprofundar o uso de threads.)

SCHILDT, Herbert. Java: A Referência Completa. 10. ed. Porto Alegre: Bookman, 2018. (Um guia de referência abrangente que detalha a biblioteca de concorrência de Java, incluindo o uso de Thread, Runnable e sincronização.)

APÊNDICE: Repositório do Código Fonte

As atividades possuem material exclusivo do portfólio que se encontra no GitHub para download.

https://github.com/allas-amk/portfolio_POO_projeto_II