Multithreading e Concorrência em Java: Uma Explicação Aprofundada

(1) Explicação Progressiva dos Fundamentos:

1. Introdução: Threads e Processos:

Em um sistema operacional, um **processo** é uma instância de um programa em execução. Cada processo tem seu próprio espaço de memória isolado. Já uma **thread** (linha de execução) é uma unidade básica de execução dentro de um processo. Um único processo pode conter múltiplas threads que compartilham o mesmo espaço de memória do processo.

A multithreading permite que um programa execute várias tarefas concorrentemente, aproveitando melhor os recursos do sistema, especialmente em sistemas com múltiplos núcleos de processamento. Isso pode levar a aplicações mais responsivas (por exemplo, uma interface gráfica que não trava enquanto realiza uma operação pesada em segundo plano) e a um melhor desempenho geral.

2. Criação de Threads:

Existem duas maneiras principais de criar threads em Java:

- **Implementando a interface** `Runnable`**:** Esta é a abordagem mais comum e flexível. Você cria uma classe que implementa a interface `Runnable`, que possui um único método: `run()`. Este método contém o código que a thread irá executar. Em seguida, você cria um objeto `Thread` passando uma instância da sua classe `Runnable` para o construtor do `Thread`. Finalmente, você inicia a thread chamando o método `start()` do objeto `Thread`. O método `start()` cria uma nova thread e faz com que o método `run()` da sua classe `Runnable` seja executado nessa nova thread.

- **Estendendo a classe** `Thread`**:** Você pode criar uma classe que herda da classe `Thread` e sobrescrever o método `run()`. Esta abordagem é menos flexível, pois sua classe já estará herdando de `Thread`, limitando a possibilidade de herdar de outra classe. Novamente, para iniciar a thread, você cria uma instância da sua classe e chama o método `start()`.

3. Sincronização:

Em um ambiente multithreaded, várias threads podem acessar e modificar recursos compartilhados (como variáveis e objetos). Isso pode levar a problemas como **condições de corrida (race conditions)**, onde o resultado da computação depende da ordem em que as threads são executadas. A

sincronização é o mecanismo usado para controlar o acesso a recursos compartilhados por múltiplas threads, garantindo que apenas uma thread acesse o recurso em um determinado momento.

- **Palavra-chave `synchronized`:** Em Java, a palavra-chave `synchronized` pode ser usada para criar blocos ou métodos sincronizados.
    - **Métodos Sincronizados:** Ao declarar um método como `synchronized`, você está garantindo que apenas uma thread pode executar esse método em um determinado objeto por vez. Quando uma thread entra em um método sincronizado, ela adquire um **lock intrínseco** (também conhecido como monitor) no objeto. Outras threads que tentarem entrar no mesmo método sincronizado do mesmo objeto terão que esperar até que a primeira thread libere o lock.
    - **Blocos Sincronizados:** Você também pode sincronizar apenas uma parte de um método usando um bloco `synchronized`. Você precisa especificar o objeto cujo lock será usado para sincronizar o bloco. Isso é útil quando apenas uma parte do método precisa de acesso exclusivo a um recurso compartilhado.
- `java.util.concurrent.locks`**:** O pacote `java.util.concurrent.locks` oferece uma API mais flexível e poderosa para sincronização através da interface `Lock` e suas implementações, como `ReentrantLock`. As locks fornecem funcionalidades adicionais em comparação com os locks intrínsecos, como a capacidade de tentar adquirir um lock (`tryLock()`), esperar por um lock com um limite de tempo e interromper uma thread enquanto ela está esperando por um lock.

4. Deadlock e Livelock:

- **Deadlock (Impasse):** Um deadlock ocorre quando duas ou mais threads ficam bloqueadas indefinidamente, esperando por um recurso que está sendo mantido por outra thread na mesma cadeia de espera. Para que um deadlock ocorra, geralmente quatro condições precisam estar presentes:
    - **Exclusão Mútua:** Apenas uma thread pode usar um recurso por vez.
    - **Espera e Retenção:** Uma thread mantém um recurso enquanto espera por outro recurso.
    - **Não Preempção:** Um recurso só pode ser liberado voluntariamente pela thread que o está mantendo.
    - **Espera Circular:** Existe uma cadeia de threads onde cada thread está esperando por um recurso mantido pela próxima thread na cadeia.
- **Livelock (Bloqueio Vivo):** Um livelock é semelhante ao deadlock, mas em vez de ficarem bloqueadas, as threads ficam constantemente tentando acessar os recursos, mas falham repetidamente devido às

ações de outras threads. As threads não ficam bloqueadas, mas também não conseguem progredir. Imagine duas pessoas tentando passar por uma porta estreita, cada uma se movendo para o lado para deixar a outra passar, mas acabando por se moverem em sincronia e bloqueando uma à outra indefinidamente.

5. Conceitos de Concorrência:

- **Atomicidade:** Uma operação é atômica se ela parece ser executada como uma única unidade indivisível. Isso significa que a operação é concluída completamente ou não é concluída de forma alguma, sem ser interrompida por outras threads. Operações simples como a leitura ou escrita de variáveis primitivas geralmente são atômicas, mas operações mais complexas podem precisar de sincronização para garantir a atomicidade.

- **Visibilidade:** A visibilidade garante que as alterações feitas por uma thread a uma variável compartilhada sejam visíveis para outras threads. Em sistemas com múltiplos processadores, cada thread pode ter sua própria cópia local da variável (cache). Sem mecanismos de sincronização adequados (como a palavra-chave `volatile` ou locks), as alterações feitas por uma thread podem não ser imediatamente visíveis para outras threads.

- **Ordenação:** A ordenação se refere à ordem em que as instruções são executadas. O compilador e o processador podem reordenar as instruções para otimizar o desempenho. Embora essa reordenação geralmente não cause problemas em programas single-threaded, ela pode levar a resultados inesperados em programas multithreaded se a sincronização adequada não for usada para estabelecer relações de ordem entre as operações de diferentes threads.

6. Classes do pacote `java.util.concurrent`:

O pacote `java.util.concurrent` fornece um conjunto rico de classes utilitárias para facilitar a programação concorrente em Java. Algumas das classes mais importantes incluem:

- `ExecutorService`: Uma interface que representa um executor de threads. Ela permite gerenciar um pool de threads e executar tarefas (`Runnable` ou `Callable`) de forma assíncrona. O uso de `ExecutorService` é preferível à criação e gerenciamento manual de threads, pois ele cuida de detalhes como o ciclo de vida das threads e o gerenciamento de recursos. Implementações comuns incluem `ThreadPoolExecutor` e as fábricas convenientes fornecidas pela classe `Executors` (como

`newFixedThreadPool()`, `newCachedThreadPool()`, `newSingleThreadExecutor()`).

- `Future`: Representa o resultado de uma computação assíncrona. Quando você submete uma tarefa para um `ExecutorService`, ele retorna um objeto `Future`. Você pode usar o `Future` para verificar se a tarefa foi concluída (`isDone()`), esperar pelo resultado (`get()`), e cancelar a tarefa (`cancel()`).

- `CountDownLatch`: Um auxiliar de sincronização que permite que uma ou mais threads esperem até que um conjunto de operações que estão sendo realizadas em outras threads seja concluído. Um `CountDownLatch` é inicializado com uma contagem. O método `countDown()` decrementa a contagem, e o método `await()` bloqueia a thread chamadora até que a contagem chegue a zero.

- `CyclicBarrier`: Um auxiliar de sincronização que permite que um conjunto de threads esperem umas pelas outras para atingir um ponto de barreira comum. Ao contrário do `CountDownLatch`, a barreira pode ser reutilizada após as threads serem liberadas. Você especifica o número de threads que precisam atingir a barreira ao criar o `CyclicBarrier`, e as threads chamam o método `await()` para esperar na barreira. Quando o número especificado de threads atinge a barreira, todas são liberadas.

- `Semaphore`: Um semáforo é um objeto que mantém uma contagem. Threads podem adquirir um "permit" do semáforo (decrementando a contagem) e liberar um permit (incrementando a contagem). Se a contagem for zero, uma thread que tentar adquirir um permit será bloqueada até que outro thread libere um permit. Semáforos são úteis para controlar o acesso a um número limitado de recursos.

(2) Resumo dos Principais Pontos:

- **Threads vs. Processos:** Threads são unidades de execução dentro de processos, compartilhando memória.
- **Criação de Threads:** Implementando `Runnable` ou estendendo `Thread`. Usar `start()` para iniciar.
- **Sincronização:** Controlar o acesso a recursos compartilhados.
    - `synchronized` (métodos e blocos): Locks intrínsecos.
    - `java.util.concurrent.locks.Lock`: API mais flexível (ex: `ReentrantLock`).
- **Deadlock:** Bloqueio mútuo indefinido de threads esperando por recursos.
- **Livelock:** Threads repetidamente reagem umas às outras sem progredir.

- **Conceitos de Concorrência:**
  - **Atomicidade:** Operação indivisível.
  - **Visibilidade:** Alterações em variáveis compartilhadas visíveis a outras threads.
  - **Ordenação:** Ordem de execução pode ser diferente da ordem no código.
- **Pacote `java.util.concurrent`:**
  - `ExecutorService`: Gerenciamento de pool de threads.
  - `Future`: Resultado de computação assíncrona.
  - `CountDownLatch`: Esperar até que um número de operações seja concluído.
  - `CyclicBarrier`: Esperar que um número de threads atinjam um ponto de barreira.
  - `Semaphore`: Controlar o acesso a um número limitado de recursos.

(3) Perspectivas e Conexões:

- **Aplicações Práticas:**
  - **Servidores Web:** Lidar com múltiplas requisições simultaneamente.
  - **Aplicações Desktop:** Realizar tarefas em segundo plano sem travar a interface do usuário.
  - **Processamento Paralelo:** Executar cálculos complexos em múltiplos núcleos.
  - **Sistemas de Mensageria:** Processar mensagens de forma concorrente.
  - **Jogos:** Gerenciar diferentes aspectos do jogo simultaneamente (IA, gráficos, física).
- **Conexões com Outras Áreas da Computação:**
  - **Sistemas Operacionais:** Gerenciamento de processos e threads.
  - **Arquitetura de Computadores:** Paralelismo em nível de hardware (múltiplos núcleos).
  - **Bancos de Dados:** Concorrência no acesso e manipulação de dados.
  - **Computação Distribuída:** Coordenação de tarefas em múltiplos nós.
  - **Inteligência Artificial e Machine Learning:** Treinamento de modelos em paralelo.

(4) Materiais Complementares Confiáveis:

- **Documentação Oficial da Oracle Java:** Seção sobre concorrência. (https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html)[1]

- **Livro "Java Concurrency in Practice" de Brian Goetz et al.:** Um livro fundamental sobre o tema.
- **Artigos e tutoriais online em sites como Baeldung (https://www.baeldung.com/java-concurrency) e GeeksforGeeks (https://www.geeksforgeeks.org/multithreading-in-java/).**
- **Cursos online especializados em plataformas como Coursera, Udemy e edX.**

(5) Exemplos Práticos:

```
Java

import java.util.concurrent.*;

import java.util.concurrent.locks.ReentrantLock;



public class MultithreadingExemplos {



    // Implementando Runnable

    static class MeuRunnable implements Runnable {

        private String nome;

        public MeuRunnable(String nome) { this.nome = nome; }

        @Override

        public void run() {

            for (int i = 0; i < 5; i++) {

                System.out.println("Thread Runnable " + nome + ": " + i);

                try { Thread.sleep(100); } catch (InterruptedException e)
{}

            }
```

```
        }
    }

    // Estendendo Thread
    static class MinhaThread extends Thread {
        private String nome;
        public MinhaThread(String nome) { this.nome = nome; }
        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                System.out.println("Thread Estendida " + nome + ": " + i);
                try { Thread.sleep(150); } catch (InterruptedException e)
{}
            }
        }
    }

    // Sincronização com synchronized
    static class ContadorSync {
        private int count = 0;
        public synchronized void incrementar() { count++; }
        public int getCount() { return count; }
```

```
    }

    // Sincronização com Lock
    static class ContadorLock {
        private int count = 0;
        private ReentrantLock lock = new ReentrantLock();
        public void incrementar() {
            lock.lock();
            try { count++; } finally { lock.unlock(); }
        }
        public int getCount() { return count; }
    }

    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        // Criação de threads
        new Thread(new MeuRunnable("A")).start();
        new MinhaThread("B").start();

        // Sincronização com synchronized
        ContadorSync contadorSync = new ContadorSync();
```

```
        Thread t1 = new Thread(() -> { for (int i = 0; i < 1000; i++)
contadorSync.incrementar(); });

        Thread t2 = new Thread(() -> { for (int i = 0; i < 1000; i++)
contadorSync.incrementar(); });

        t1.start();

        t2.start();

        t1.join();

        t2.join();

        System.out.println("Contador Sync: " + contadorSync.getCount());



        // Sincronização com Lock

        ContadorLock contadorLock = new ContadorLock();

        Thread t3 = new Thread(() -> { for (int i = 0; i < 1000; i++)
contadorLock.incrementar(); });

        Thread t4 = new Thread(() -> { for (int i = 0; i < 1000; i++)
contadorLock.incrementar(); });

        t3.start();

        t4.start();

        t3.join();

        t4.join();

        System.out.println("Contador Lock: " + contadorLock.getCount());



        // ExecutorService
```

```
        ExecutorService executor = Executors.newFixedThreadPool(2);

        Future<String> future = executor.submit(() -> {

            Thread.sleep(2000);

            return "Tarefa concluída!";

        });

        System.out.println("Tarefa submetida...");

        // System.out.println("Resultado da tarefa: " + future.get()); //
Espera o resultado

        executor.shutdown();



        // CountDownLatch

        CountDownLatch latch = new CountDownLatch(3);

        new Thread(() -> { System.out.println("Tarefa 1 concluída");
latch.countDown(); }).start();

        new Thread(() -> { System.out.println("Tarefa 2 concluída");
latch.countDown(); }).start();

        new Thread(() -> { System.out.println("Tarefa 3 concluída");
latch.countDown(); }).start();

        latch.await();

        System.out.println("Todas as tarefas foram concluídas!");



        // CyclicBarrier

        CyclicBarrier barrier = new CyclicBarrier(2);
```

```
        Thread worker1 = new Thread(() -> {

            try { System.out.println("Worker 1 atingiu a barreira");
barrier.await(); System.out.println("Worker 1 passou a barreira"); } catch
(Exception e) {}

        });

        Thread worker2 = new Thread(() -> {

            try { System.out.println("Worker 2 atingiu a barreira");
barrier.await(); System.out.println("Worker 2 passou a barreira"); } catch
(Exception e) {}

        });

        worker1.start();

        worker2.start();



        // Semaphore

        Semaphore semaphore = new Semaphore(2); // Permite até 2 threads
acessarem

        new Thread(() -> {

            try { semaphore.acquire(); System.out.println("Thread S1
adquiriu o semáforo"); Thread.sleep(1000); semaphore.release();
System.out.println("Thread S1 liberou o semáforo"); } catch
(InterruptedException e) {}

        }).start();

        new Thread(() -> {

            try { semaphore.acquire(); System.out.println("Thread S2
adquiriu o semáforo"); Thread.sleep(1000); semaphore.release();
System.out.println("Thread S2 liberou o semáforo"); } catch
(InterruptedException e) {}
```

```
        }).start();

        new Thread(() -> {

            try { semaphore.acquire(); System.out.println("Thread S3
adquiriu o semáforo"); Thread.sleep(1000); semaphore.release();
System.out.println("Thread S3 liberou o semáforo"); } catch
(InterruptedException e) {}

        }).start();

    }

}
```

(6) Metáforas e Pequenas Histórias para Memorização:

- **Threads como Funcionários em um Escritório:** Imagine um escritório (processo) com vários funcionários (threads) trabalhando na mesma sala (compartilhando memória). Cada funcionário pode realizar uma tarefa diferente simultaneamente.

- **Sincronização como um Semáforo no Banheiro:** Imagine um banheiro (recurso compartilhado) com um semáforo (sincronização). Apenas uma pessoa (thread) pode entrar no banheiro quando o semáforo está verde. As outras pessoas (threads) precisam esperar até que o semáforo fique verde novamente.

- **Deadlock como Duas Pessoas Presas em uma Porta Giratória:** Imagine duas pessoas (threads) tentando passar por uma porta giratória. A primeira pessoa precisa que a segunda se mova para poder passar, e a segunda pessoa precisa que a primeira se mova para poder passar. Ambas ficam bloqueadas, esperando uma pela outra.

- **Atomicidade como uma Transação Bancária:** Uma transação bancária (operação atômica) deve ser concluída completamente (débito e crédito) ou não ser concluída de forma alguma. Você não quer que o dinheiro seja debitado de uma conta, mas não creditado na outra.

- **Visibilidade como um Quadro de Avisos:** Imagine um quadro de avisos (memória compartilhada). Se um funcionário (thread) escreve uma mensagem no quadro, os outros funcionários (threads) precisam conseguir ver essa mensagem.

- `ExecutorService` **como um Gerente de Projetos:** Um gerente de projetos (`ExecutorService`) recebe tarefas e as distribui para uma equipe de trabalhadores (pool de threads), gerenciando o fluxo de trabalho.

- `Future` **como um Ticket de Pedido:** Quando você faz um pedido em um restaurante, você recebe um ticket (`Future`). Esse ticket representa o seu pedido (tarefa) que ainda está sendo preparado. Você pode usar o ticket para verificar se o pedido está pronto (`isDone()`) e para receber o seu prato (resultado `get()`).

- `CountDownLatch` **como a Contagem Regressiva para um Lançamento:** Imagine uma contagem regressiva para o lançamento de um foguete. Várias verificações (tarefas) precisam ser concluídas antes que o lançamento possa ocorrer. O `CountDownLatch` garante que o foguete só seja lançado (a thread principal continue) depois que todas as verificações forem feitas.

- `CyclicBarrier` **como um Ponto de Encontro para um Grupo de Viajantes:** Imagine um grupo de viajantes que se encontram em diferentes pontos de uma cidade e combinam de se encontrar em um ponto específico (barreira) antes de seguirem juntos para o próximo destino.

- `Semaphore` **como Vagas de Estacionamento:** Imagine um estacionamento com um número limitado de vagas (permits). Os carros (threads) podem entrar no estacionamento se houver vagas disponíveis (adquirir um permit) e precisam liberar a vaga ao sair (liberar um permit). Se todas as vagas estiverem ocupadas, os carros precisam esperar.