# Resumo Completo de Multithreading em Java

Este documento serve como um guia consolidado sobre os conceitos de programação concorrente em Java, com base em nossa conversa.

## 1. Fundamentos do Multithreading

### O que é Multithreading?

Multithreading é a capacidade de um programa executar múltiplos fluxos de execução, conhecidos como **threads**, simultaneamente dentro de um único processo. Isso permite que um programa execute várias tarefas ao mesmo tempo, melhorando a eficiência e a responsividade.

### Por que usar Multithreading?

* **Eficiência**: Aproveita melhor o hardware moderno (processadores com múltiplos núcleos) para executar tarefas em paralelo.
* **Responsividade**: Mantém a interface do usuário ágil, mesmo quando o programa está executando tarefas longas em segundo plano.
* **Escalabilidade**: Facilita a adaptação do programa para lidar com um volume maior de trabalho.

### Concorrência vs. Paralelismo

* **Concorrência**: Várias tarefas progridem ao longo do tempo, disputando os mesmos recursos e se alternando na execução.
* **Paralelismo**: Várias tarefas são executadas no mesmo instante, o que requer hardware com múltiplos núcleos.

## 2. A Vida de uma Thread: O Ciclo de Vida

Uma thread passa por diferentes estados durante sua existência:

* **New (Iniciada)**: A thread foi criada como um objeto, mas ainda não foi iniciada.
* **Runnable (Pronta)**: O método start() foi chamado. A thread está na fila, pronta para ser executada e aguardando sua vez de usar a CPU.
* **Blocked (Bloqueada)**: A thread é parada temporariamente porque está tentando acessar um recurso que está sendo usado por outra thread (ex: um bloco synchronized).
* **Waiting (Esperando)**: A thread se colocou em espera intencionalmente, aguardando um sinal específico de outra thread para continuar.
* **Terminated (Terminada)**: A execução do método run() foi concluída. A thread não pode ser executada novamente.

## 3. Criando e Iniciando Threads

Existem duas maneiras principais, mas a primeira é a mais recomendada.

### 1. Implementando a interface Runnable (Forma Preferida)

Esta abordagem é mais flexível, pois sua classe pode herdar de outra classe se necessário.

// Define a tarefa a ser executada  
public class MeuRunnable implements Runnable {  
 public void run() {  
 System.out.println("Executando em uma nova thread!");  
 }  
}  
  
// Cria e inicia a thread  
Thread t = new Thread(new MeuRunnable());  
t.start(); // Inicia a nova thread e chama o método run()

### A Diferença Crucial: start() vs. run()

* t.start(): Cria uma nova thread e a coloca no estado **Runnable**. A JVM então agenda a execução do método run() nessa nova thread.
* t.run(): **Não cria uma nova thread**. Apenas executa o método run() na thread atual, como uma chamada de método comum.

## 4. O Desafio da Sincronização

### Condições de Corrida (Race Conditions)

Ocorrem quando duas ou mais threads acessam e modificam um recurso compartilhado, e o resultado final depende da ordem imprevisível em que as threads são executadas. Isso pode levar a dados inconsistentes.

### Solução 1: A Palavra-chave synchronized

Funciona como um "cadeado" (lock) para proteger um bloco de código ou um método. Apenas uma thread pode obter o lock por vez.

public class Contador {  
 private int c = 0;  
  
 // Apenas uma thread pode executar este método por vez  
 public synchronized void incrementar() {  
 c++;  
 }  
}

É uma boa prática sincronizar apenas a **seção crítica** do código para não prejudicar a performance.

### Solução 2: Classes Atômicas

Oferecem uma alternativa mais eficiente para operações simples (como incrementar um número) sem o uso de synchronized. Elas garantem que a operação seja **indivisível (atômica)**.

import java.util.concurrent.atomic.AtomicInteger;  
  
public class ContadorAtomico {  
 private AtomicInteger c = new AtomicInteger(0);  
  
 public void incrementar() {  
 // Esta operação é thread-safe e indivisível  
 c.incrementAndGet();  
 }  
}

## 5. Coleções em Ambientes Multithread

Coleções padrão como ArrayList não são seguras para threads.

* **Abordagem Antiga**: Collections.synchronizedList() cria um "wrapper" que bloqueia a coleção inteira a cada acesso, o que pode ser lento.
* **Abordagem Moderna**: Use as classes do pacote java.util.concurrent, como CopyOnWriteArrayList (ótima para muitas leituras) e ConcurrentHashMap (ótima para acesso concorrente em mapas).

## 6. Gerenciamento com o Executor Framework

O ExecutorService gerencia um "pool de threads" para reutilizar threads e melhorar a performance.

* **newFixedThreadPool(4)**: Cria um pool com um número fixo de 4 threads.
* **newCachedThreadPool()**: Cria um pool dinâmico que cria e remove threads conforme a necessidade.

Para obter resultados de tarefas, use Callable (que retorna um valor) em conjunto com Future (que representa o resultado futuro da computação).

ExecutorService executor = Executors.newFixedThreadPool(2);  
Callable<Integer> tarefa = () -> {  
 // Simula um cálculo demorado  
 Thread.sleep(1000);  
 return 123;  
};  
  
Future<Integer> futuro = executor.submit(tarefa);  
  
// O método get() bloqueia até o resultado estar pronto  
Integer resultado = futuro.get();  
System.out.println("O resultado é: " + resultado);  
  
executor.shutdown();

## 7. Agendamento e Coordenação de Threads

### ScheduledExecutorService

Permite agendar tarefas para execução futura ou repetida.

### Ferramentas de Coordenação

* **CyclicBarrier**: Um "ponto de encontro". Bloqueia um grupo de threads até que todas cheguem à barreira, sendo ideal para tarefas que precisam ser sincronizadas em ciclos.
* **CountDownLatch**: Uma "catraca de contagem regressiva". Uma thread espera (await()) até que um número N de outras tarefas seja concluído (countDown()). É para eventos únicos.
* **Semaphore**: Um "controlador de vagas". Limita o número de threads que podem acessar um recurso ou executar uma seção de código simultaneamente.