LP Orientada a Objetos II

Aula 04

Diego Addan

DS142 - UFPR - 2023

Para hoje

Programação Concorrente

Threads

Exercícios

Anteriormente

1 - Utilizando Streams, crie um programa em Java que implemente a função **uniao**. A função recebe como parâmetros dois arrays de inteiros e retorna um novo array contendo a união de v1 e v2.
Por exemplo, se v1 = {11, 13, 45, 7} e v2 = {24, 4, 16, 81, 10, 12}, v3 irá conter {11, 13, 45, 7, 24, 4, 16, 81, 10, 12}.

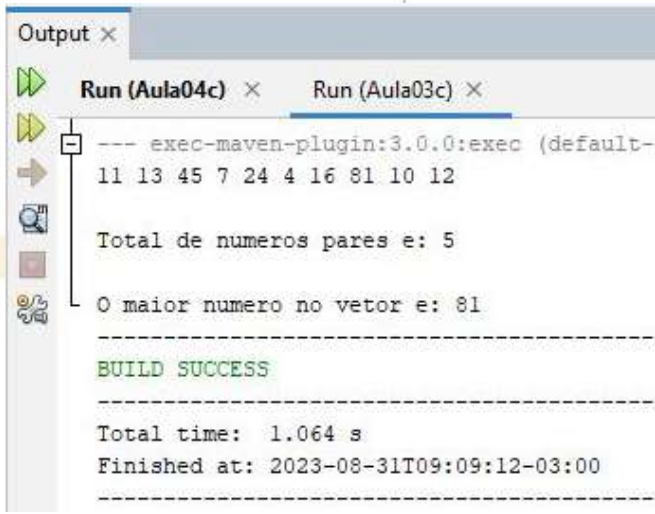
2 - Usando Streams, faça um sistema que encontre:
a) totalPares: a função recebe como parâmetro um array de inteiros e retorna um número inteiro indicando o total de números pares existentes no array.
b) maiorValor: a função recebe como parâmetro um array de inteiros e retorna o maior número existente no array.

Anteriormente

```
//
int[] v1 = {11, 13, 45, 7};
int[] v2 = {24, 4, 16, 81, 10, 12};

int[] v3 = mUniao(v1, v2);      // Método de união de Arrays
Arrays.stream( array: v3).forEach(num -> System.out.print(num + " "));

System.out.println("\n\nTotal de numeros pares eh: " + mPares( v1: v3));

System.out.println("\nO maior numero no vetor eh: " + mMaior( array: v3));
```

Anteriormente

```java
public static int[] mUniao(int[] v1, int[] v2) {
    return IntStream.concat( a: Arrays.stream( array: v1),  b: Arrays.stream( array: v2))
                    .distinct()
                    .toArray();
}

public static int mPares(int[] v1) {
    return (int) IntStream.of( values: v1)
                          .filter(num -> num % 2 == 0)
                          .count();
}

public static int mMaior(int[] array) {
    return Arrays.stream(array).max().getAsInt();
}
```



Anteriormente

```java
public static int[] mUniao(int[] v1, int[] v2) {
    return IntStream.concat( a: Arrays.stream( array: v1),  b: Arrays.stream( array: v2))
                    .distinct()
                    .toArray();
}

public static int mPares(int[] v1) {
    return (int) IntStream.of( values: v1)
                          .parallel()
                          .filter(num -> num % 2 == 0)
                          .count();
}

public static int mMaior(int[] array) {
    return Arrays.stream(array).parallel().max().getAsInt(
}
```

```
11 13 45 7 24 4 16 81 10 12

Total de numeros pares eh: 5

O maior numero no vetor eh: 81
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  1.110 s
Finished at: 2023-08-31T09:15:12-03:00
------------------------------------------------------------------------
```

Threads

Java permite trabalhar de forma paralela, também conhecida por programação concorrente, através de Threads.

Subdivisões dos Processos

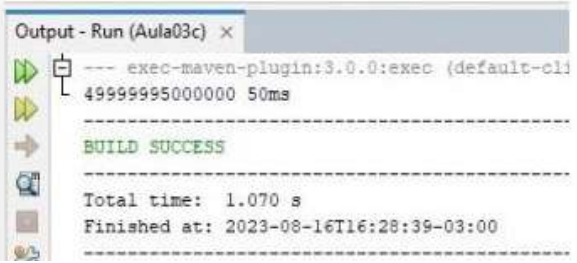
Cada processo possui diversas threads (linhas de instruções), assim nós podemos dividir partes do nosso processo (programa em Java) para trabalhar paralelamente.

Threads

Executar **1 ou mais threads** ao mesmo tempo, ou seja, 1 ou mais **procedimentos internos** do programa ao mesmo tempo através de processadores lógicos.

Stream Paralelo

```
res = LongStream.range(1L, num)
        .parallel()
        .reduce(0L, Long::sum);
```



Daemon

User

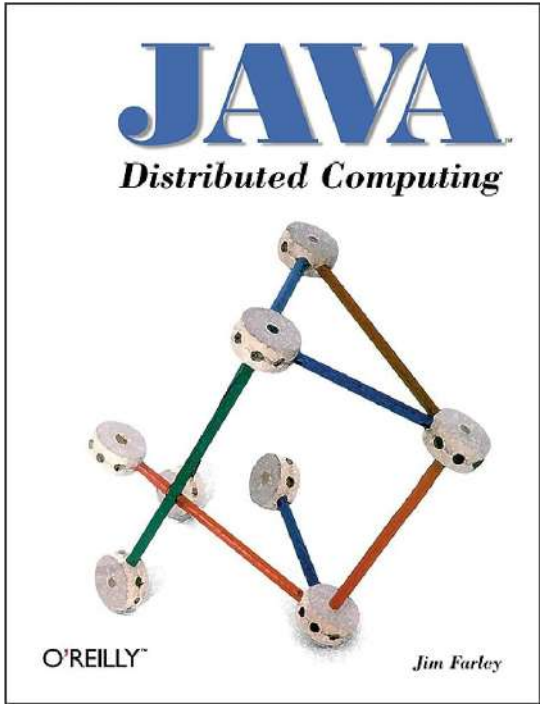
Programação Paralela e Distribuída em Java

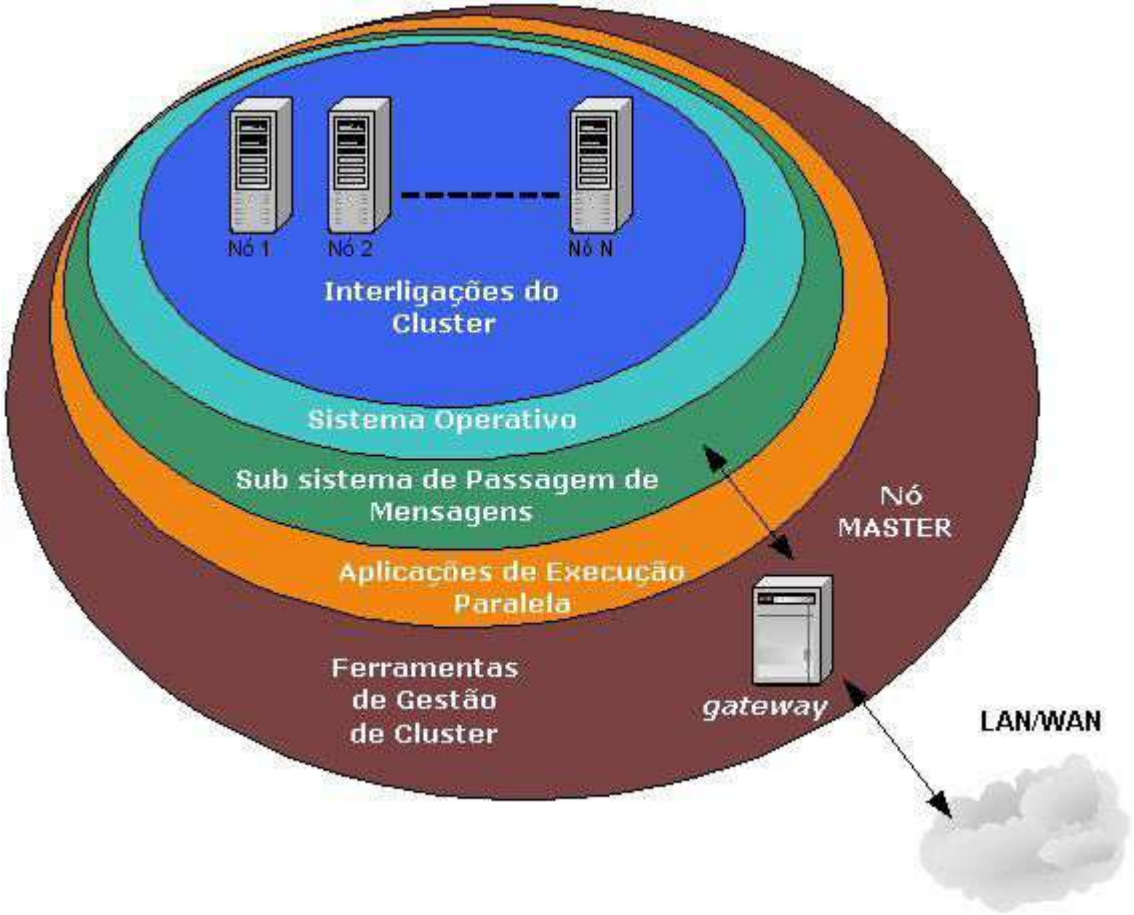
Programação Paralela: estratégia que consiste na execução simultânea de partes distintas de uma mesma aplicação, minimizando tempo na obtenção de resultados de tarefas grandes e complexas

Programação Distribuída: execução de aplicações cooperantes em computadores diferentes interligados por uma rede física ou lógica. Geralmente utilizada em Sistemas Multimídia e em Computação Móvel

Programação Paralela e Distribuída em Java

Programação Distribuída





Programação Paralela e Distribuída em Java

Programação Paralela

Decomposição funcional: diversos programas menores que serão distribuídos entre os processadores para execução simultânea
Decomposição dos dados: grupos de valores que serão distribuídos entre os processadores que executarão simultaneamente um mesmo programa, então se tem a chamada

Threads e Multithreading

Programação Paralela e Distribuída em Java

Sincronização

Quando threads são disparadas dentro de uma mesma aplicação, é necessário sincronizá-las para evitar que dados compartilhados se tornem inconsistentes ou então que essas threads atuem em momentos errados.

Existem duas formas de sincronização:

- **Competição**: quando duas ou mais threads competem pelo mesmo recurso compartilhado.
- **Cooperação**: comunicação entre Threads para que uma atue num momento específico que depende de uma ação ou estado da outra.

Threads

Temos uma classe Thread que nos permite tratar processos internos como tarefas

Ex: System.out.println(Thread.currentThread().getName());

Neste caso podemos criar uma classe derivada da Threads e trabalhar com seus recursos:

```
class Thread01 extends Thread{
    private char c;
    public Thread01(char c){
        this.c = c;
    }
    @Override
    public void run(){
    System.out.println( x: Thread.currentThread().getName());
        for(int i=0; i <200; i++){
            System.out.print(c);
        }
    }
}
```

Threads

Ex

```java
public class Aula04 {
        public static void main(String[] args) {
            Thread01 t1 = new Thread01( c: 'A');
            Thread01 t2 = new Thread01( c: 'B');
            Thread01 t3 = new Thread01( c: 'C');
            t1.run(); t2.run(); t3.run();
        }
```

com.mycompany.tads1.Aula04

Output - Run (Aula04)

```
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.0.0:exec (default-cli) @ tads001 ---
main
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  1.180 s
Finished at: 2023-08-28T14:26:17-03:00
------------------------------------------------------------------------
```

Threads

Se trocarmos a chamada de run para start temos uma execução concorrente de processo:

```
    public static void main(String[] args) {
        Thread01 t1 = new Thread01( c: 'A');
        Thread01 t2 = new Thread01( c: 'B');
        Thread01 t3 = new Thread01( c: 'C');
        t1.start(); t2.start(); t3.start();
```

com.mycompany.tads1.Aula04 > main > t2

ut - Run (Aula04)

```
Building tads001 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.0.0:exec (default-cli) @ tads001 ---
Thread-0
AAAAAAAAThread-1
BBBBBBBBBBBThread-2
CCCCCCCCCCCAABBBBBBBBBBBBBBCCCCCCAAAAAAAABBBBBBBBBCCCCCCCCCAAAAAAAABBBBBBBBBCAAAAAAAABBBBBBBBBCCCCCCCCCAAAAAAA
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  1.285 s
Finished at: 2023-08-28T14:30:35-03:00
------------------------------------------------------------------------
```

Threads

Neste caso o controle é feito pela **JVM** (caixa preta)
Utiliza os cores disponíveis e trabalha o gargalo de memória

Podemos implementar a interface **Runnable** e ter uma execução semelhante:

```
class Thread01 implements Runnable{
    private char c;
    public Thread01(char c){
        this.c = c;
    }
    @Override
    public void run(){
    System.out.println( x: Thread.currentThread().getName());
        for(int i=0; i <200; i++){
            System.out.print(c);
        }
```

```
--- exec-maven-plugin:3.0.0:exec (default-cli) @ tads
Thread-0
AAAAAAAThread-2
CCCCCCCCCCCCThread-1
BBBBBBBBBBBBAAAAAAAAAACCCCCCCCCCBBBBBBBAAAAAAACCCCCCBBB
------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------
Total time:  1.153 s
Finished at: 2023-08-28T14:43:31-03:00
------------------------------------------------------------
```

Threads

- Quando você define um thread estendendo a classe Thread, você deve **sobrescrever** o método **run( )** na classe Thread.
- Quando você implementa uma interface Runnable, você precisa **implementar** o único método **run( )** da interface Runnable

Base para Comparação	Fio	Runnable
Basic	Cada thread cria um objeto exclusivo e é associado a ele.	Vários segmentos compartilham os mesmos objetos.
Memória	Como cada thread cria um objeto único, mais memória é necessária.	Como vários segmentos compartilham o mesmo objeto, menos memória é usada.
Estendendo	Em Java, herança múltipla não é permitida, portanto, depois que uma classe extende a classe Thread, ela não pode estender nenhuma outra classe.	Se uma classe define thread implementando a interface Runnable, tem a chance de estender uma classe.
Usar	Um usuário deve estender a classe de encadeamento somente se desejar substituir os outros métodos na classe Thread.	Se você deseja apenas se especializar em executar o método, a execução do Runnable é a melhor opção.

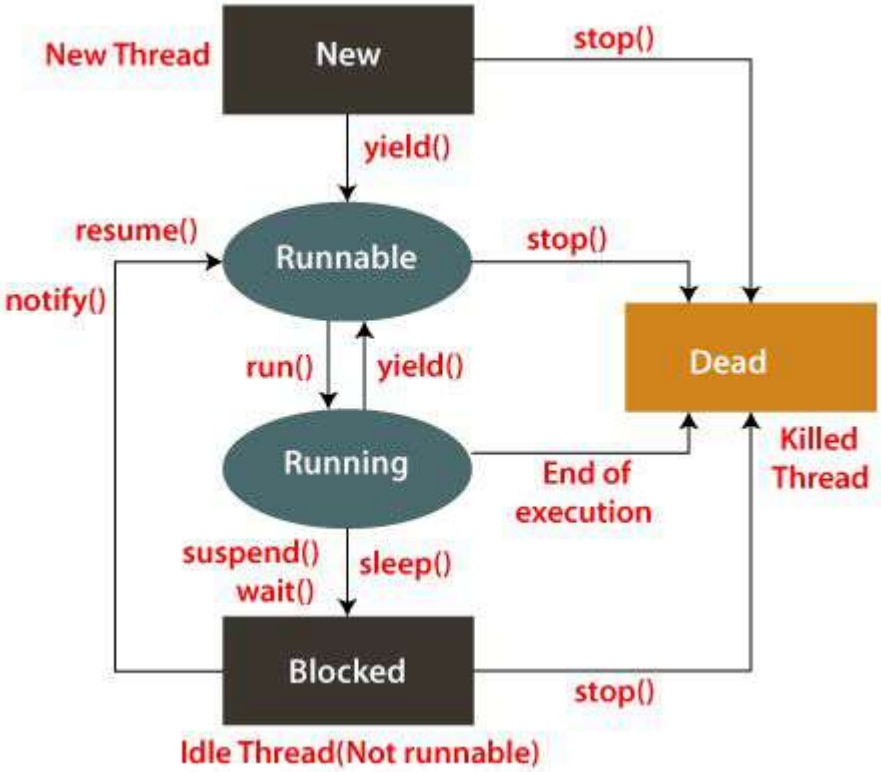
Threads

Com multithreading adquirimos vários benefícios, tais como:

- Threads compartilham o mesmo espaço de endereço e, portanto, podem compartilhar dados e código;
- Trocas de contexto entre threads são geralmente menos custosas do que entre processos;
- Custos de intercomunicação entre as threads são relativamente menores que a dos processos;
- Threads permitem diferentes tarefas a serem realizadas simultaneamente;
- Cada Thread possui uma prioridade, que define se ela será executada antes ou depois de outra.

Threads

- As Threads especificam pontos de controle para o interpretador onde a execução pode ser dinâmica
- Ainda é modular (thread Main)
- **Schedule** decide a prioridade de execução

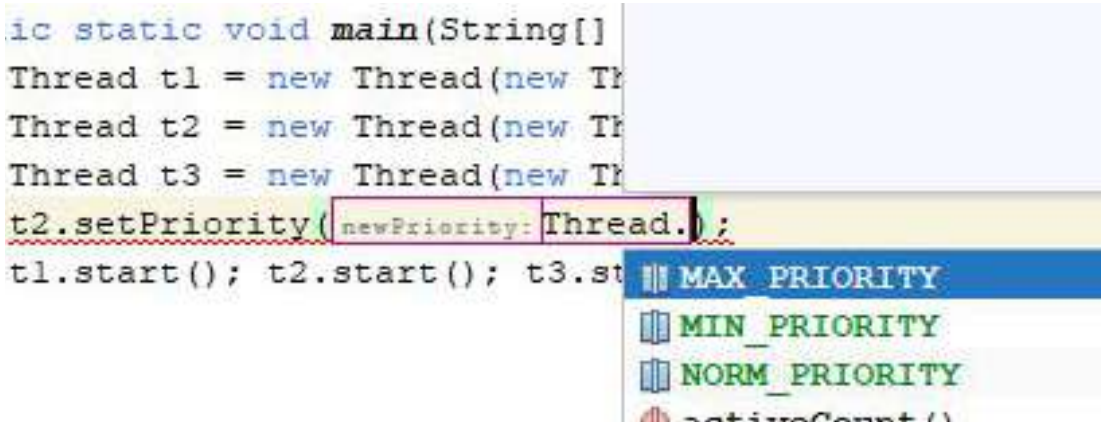


Threads

Prioridade:

Uma Thread java pode ter a Prioridade definida: [1-10]

Ex: Minha_thread.setPriority(Valor)



Não funciona como Semáforo de processos!

Podemos definir também um intervalo de bloqueio de execução:

```
try {
    Thread.sleep( millis: 1000);
} catch (InterruptedException ex) {
    ex.printStackTrace();
}
```

Threads

Podemos implementar Threads com programação funcional.

Ex: **Inner Class/ Método Anônimo**

```
Thread teste = new Thread(()->{
// Codigo aqui
});
```

Desta forma podemos criar
Um processo escalonável
Independente de modelo

```
Thread teste = new Thread(() -> {
    for (int i = 0; i < 200; i++) {
        System.out.println(i + ", ");
    }
});
teste.start();
```

Threads

Diferente do Sleep, onde a Thread fica em estado suspenso, o comando Yield serve para voltar a condição de Runnable (na fila, deixando o escalonador apontar as próximas Threads)

Ex: Dentro do método thread.yield();

Uma forma de definir que um processo deve ser finalizado antes de seguir é utilizando o Join

Ex: thread01.start();

```
thread01.join();

thread02.start();
```

Threads

Ex: Fazer um processo de cálculo aritmético em três Threads simultâneos

```
class Tarefa extends Thread {

    private final long valorInicial;
    private final long valorFinal;
    private long total = 0;

    public Tarefa(int valorInicial, int valorFinal) {
        this.valorInicial = valorInicial;
        this.valorFinal = valorFinal;
    }
    public long getTotal() {
        return total;
    }
    /*
     Este método se faz necessário para que possamos dar start()
     e iniciar a tarefa em paralelo
     */
    @Override
    public void run() {
        for (long i = valorInicial; i <= valorFinal; i++) {
            total += i;
        }
    }
```

Threads

Ex: Fazer um processo de cálculo aritmético em três Threads simultâneos

```
public static void main(String[] args) {
  //cria 3 tarefas
  Tarefa t1 = new Tarefa( valorInicial: 0,  valorFinal: 1000);
  t1.setName( name: "Tarefa1");
  Tarefa t2 = new Tarefa( valorInicial: 1001,  valorFinal: 2000);
  t2.setName( name: "Tarefa2");
  Tarefa t3 = new Tarefa( valorInicial: 2001,  valorFinal: 3000);
  t3.setName( name: "Tarefa3");

  //inicia a execução paralela das 3 tarefas, iniciando 3 novas threads no programa
  t1.start();
  t2.start();
  t3.start();

  //aguarda a finalização das tarefas
  try {
      t1.join();
      t2.join();
      t3.join();
  } catch (InterruptedException ex) {
      ex.printStackTrace();
  }

  //Exibimos o somatório dos totalizadores de cada Thread
  System.out.println("Total: " + (t1.getTotal() + t2.getTotal() + t3.getTotal()));
```

Threads

Este exemplo foi uma forma simples e prática de paralelizarmos tarefas em um programa, mas de forma estática, pois a quantidade de threads está prefixada.

Seguindo a teoria da programação concorrente, podemos tornar tudo isso dinâmico, criando a quantidade de threads ideal para o número de núcleos de processamento existentes no ambiente no qual está sendo executado o programa. O código a seguir mostra como conseguir esta informação:

```
//armazena a quantidade de núcleos de processamento disponíveis
int numThreads = Runtime.getRuntime().availableProcessors();
Collection<Tarefa> threads = new ArrayList<>();
//cria threads conforme a quantidade de núcleos
for (int i = 1; i <= numThreads; i++) {
                Tarefa thread = new Tarefa();
                thread.setName("Thread "+i);
        threads.add(thread);      }
```

Threads

Ainda no sincronismo de processos, além do join, que é executado na instância, podemos definir uma regra para execução sem concorrência no modelo com o atributo synchronized:

Ex: **Class Thread_Exemplo{**

```
        ....
  private synchronized void metodo( ){ ... }
}
```

Desta forma o método vai ser executado sempre de forma prioritária por chamada, independente de quantas threads estão em execução (**locked**)

Executa de forma atômica!

Threads e Concorrência

A maneira que vimos até agora é manual. Existe um pacote que ajuda na abstração ao trabalharmos com Threads

Vamos imaginar um problema: Temos um spawn de clientes e outro de servidores. Precisamos que o sistema saiba como está a distribuição destes objetos entre as Threads.

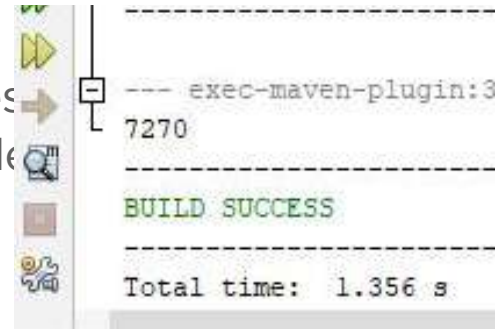
```
class Contador{
    public int count;
    void incrementa(){
        count = count + 1;
    }

    public int getCount() {
        return count;
    }
}
```

```
Contador ct = new Contador();
Runnable r = () ->{
 for (int i = 0; i < 5000; i++){
     ct.incrementa();
}
};
Thread tr1 = new Thread( task: r); Thread tr2 = new Thread( task: r);
tr1.start(); tr2.start(); tr1.join(); tr2.join();
System.out.println( x: String.valueOf( i: ct.getCount()));
```

Threads e Concorrência

A maneira que vimos até agora é manual. Existe um pacote que ajuda na abstração ao trabalharmos com Threads

Vamos imaginar um problema: Temos um spawn de clientes
Precisamos que o sistema saiba como está a distribuição de
Threads.




```
class Contador{
    public int count;
    void incrementa(){
        count = count + 1;
    }

    public int getCount() {
        return count;
    }
}
```

```
Contador ct = new Contador();
Runnable r = () ->{
 for (int i = 0; i < 5000; i++){
     ct.incrementa();
}
};
Thread tr1 = new Thread( task: r); Thread tr2 = new Thread( task: r);
tr1.start(); tr2.start(); tr1.join(); tr2.join();
System.out.println( x: String.valueOf( i: ct.getCount()));
```

Threads e Concorrência

Incluímos a sincronização para que a contagem seja validada (dessa forma o valor fica correto)

```
synchronized void incrementa(){
        count = count + 1;}
```

O problema aqui é que a execução passa a ser linear, logo a performance cai

Então podemos utilizar o pacote concurrent e a classe AtomicInteger

```
class Contador{
    public int count;
    private AtomicInteger atmc = new AtomicInteger();
    void incrementa(){
        count = count + 1;
        atmc.incrementAndGet();
    }
```

Threads e Concorrência

```
class Contador{
    public int count;
    private AtomicInteger atmc = new AtomicInteger();
    void incrementa(){
        count = count + 1;
        atmc.incrementAndGet();
    }
```

```
79          Thread tr1 = new Thread( task: r); Thread tr2 = new Thread( task: r);
80          tr1.start(); tr2.start(); tr1.join(); tr2.join();
81          System.out.println( x: String.valueOf( i: ct.getCount()));
82          System.out.println( x: ct.getAtmc());
83      }
```

com.mycompany.tads1.Aula04 > main >

Output - Run (Aula04) ×

```
------------------------< com.mycompany:tads001 >-------------------------
Building tads001 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.0.0:exec (default-cli) @ tads001 ---
9323
10000
------------------------------------------------------------------------
```

Threads e Concorrência

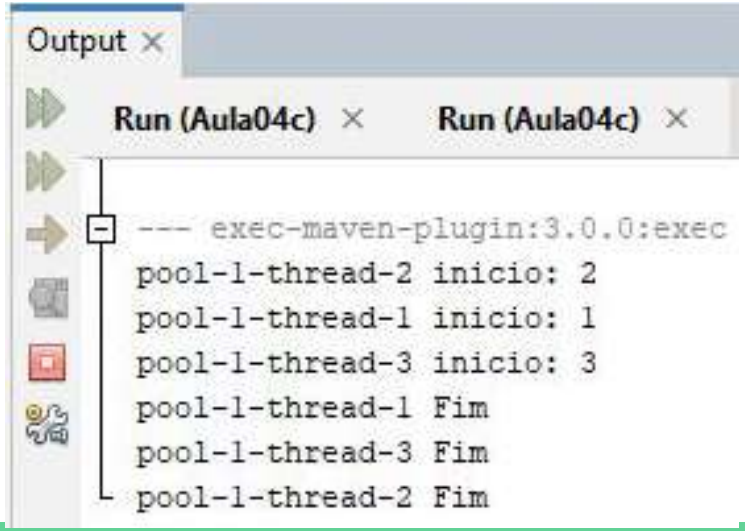
Executors: Desacopla a submissão de tarefas da execução.
Vamos criar um método para imprimir uma Thread

```
class Imprime implements Runnable{
        private final int num;
        public Imprime(int num){
            this.num = num;
        }

        @Override
        public void run(){
            System.out.printf( format: "%s inicio: %d%n",  args: Thread.currentThread().getName(),  args: num);
            try {
                TimeUnit.SECONDS.sleep( timeout: 3);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
            System.out.printf( format: "%s Fim: %d%n",  args: Thread.currentThread().getName());
        }
}
```

Threads e Concorrência

Executors: Iniciamos o Runnable em uma sequência de threads

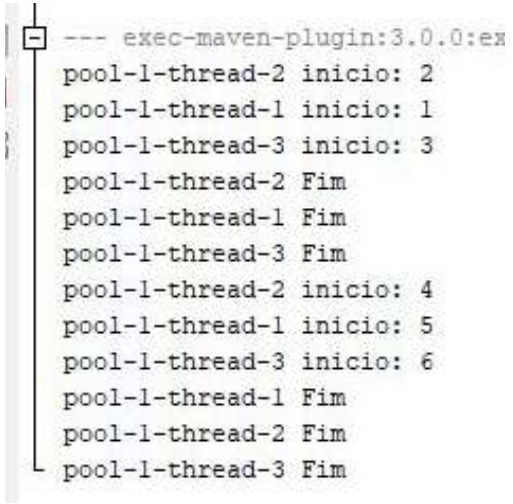
```
public static void main(String[] args) {
    ExecutorService es = Executors.newFixedThreadPool( nThreads: 3);
    es.execute(new Imprime( num: 1));
    es.execute(new Imprime( num: 2));
    es.execute(new Imprime( num: 3));
}
```



Threads e Concorrência

Executors: O interessante é que podemos instanciar mais execuções do que o número de Threads criado, e ele utilizará um dos núcleos quando ficar disponível:

```
ExecutorService es = Executors.newFixedThreadPool(3);

es.execute(new Imprime(1));

es.execute(new Imprime(2));

....

es.execute(new Imprime(6));
```



Utilizamos o shutdown para interromper as threads do executor.

```
es.shutdown();
```

Programação Paralela em Java

Existem bibliotecas externas que permitem utilizar algoritmos de programação paralela em Java

https://pcj.icm.edu.pl/


```
import org.pcj.*;

public class HelloWorld implements StartPoint {

    public static void main(String[] args) throws IOException {
    String nodesFile  = "nodes.txt";
    PCJ.executionBuilder (PcjExample.class)
                .addNodes(new File("nodes.txt"))
                .start();
    }
    }

    @Override
    public void main() throws Throwable {
                System.out.println("Hello World from PCJ Thread " + PCJ.myId()
                                    + " out of " + PCJ.threadCount() );
            }
}
```

Programação Paralela em Java

https://pcj.icm.edu.pl/

```
wrz 23, 2016 2:00:22 AM org.pcj.internal.InternalPCJ start
INFO: Starting HelloWorld with 4 threads (on 1 node)...
 Hello World from PCJ Thread 2 out of 4
 Hello World from PCJ Thread 0 out of 4
 Hello World from PCJ Thread 1 out of 4
 Hello World from PCJ Thread 3 out of 4
wrz 23, 2016 2:00:22 AM org.pcj.internal.InternalPCJ start
INFO: Completed HelloWorld with 4 threads (on 1 node) after 0h 0m 0s.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Concluindo

É possível paralelizar processos dependendo do **problema**

Limitações como JVM devem ser considerados, assim como o fluxo de compilação

Stream, Threads e Concorrência

Exercício

Escreva um programa que realize o cálculo das somas dos valores das linhas de uma matriz [n]x[n] de números inteiros e imprima o resultado total.

Faça com que o cálculo do somatório de cada linha seja realizado em paralelo utilizando **thread**.

Referências

1. DEITEL. JAVA Como Programar. 8a. ed. São Paulo: Pearson Prentice Hall, 2010.

2. JANDL JUNIOR, Peter. Java Guia do Programador. São Paulo: Novatec, 2014.

3. FREEMAN, Eric. Use a cabeça: padrões e projetos. 2. ed. rev. Rio de Janeiro: Alta Books, 2009