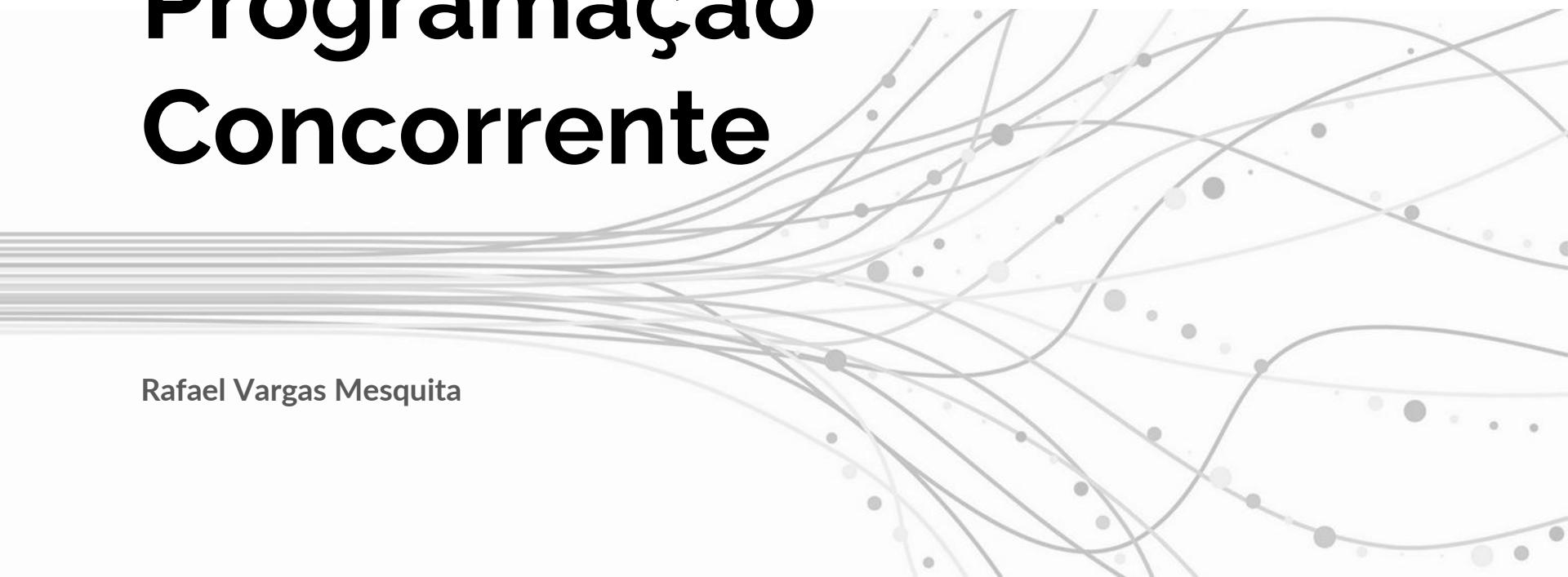




Programação Concorrente

Rafael Vargas Mesquita



ROTEIRO

Introdução

Estado de thread: ciclo de vida

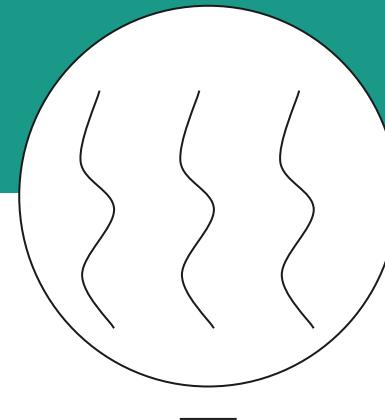
Prioridades e agendamento de threads

Criando e executando threads

Sincronização de threads

Multithreading com GUI

Programação Concorrente



Introdução

Rafael Mesquita, Prof.



Introdução

O *corpo humano* realiza uma grande variedade de operações paralelamente.

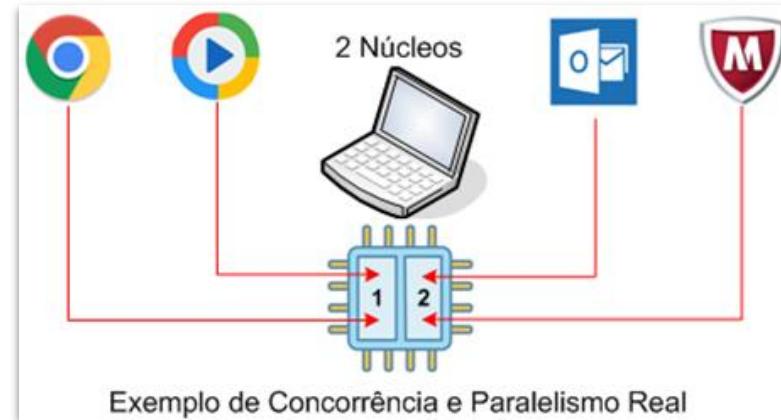
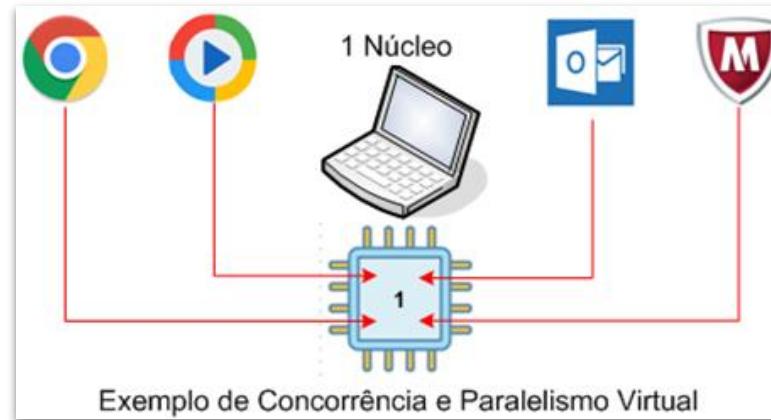


Introdução

Os computadores também realizam operações paralelamente e/ou concorrentemente.

Paralelamente: tarefas são executadas simultaneamente.

Concorrentemente: tarefas disputam o mesmo recurso.



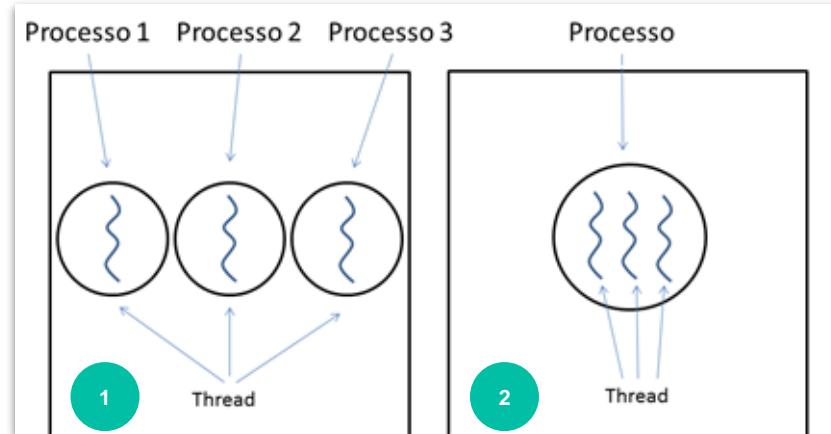
Introdução

Programa: conjunto de instruções em uma linguagem de alto nível ou de máquina.

Processo: resultado da execução do programa.

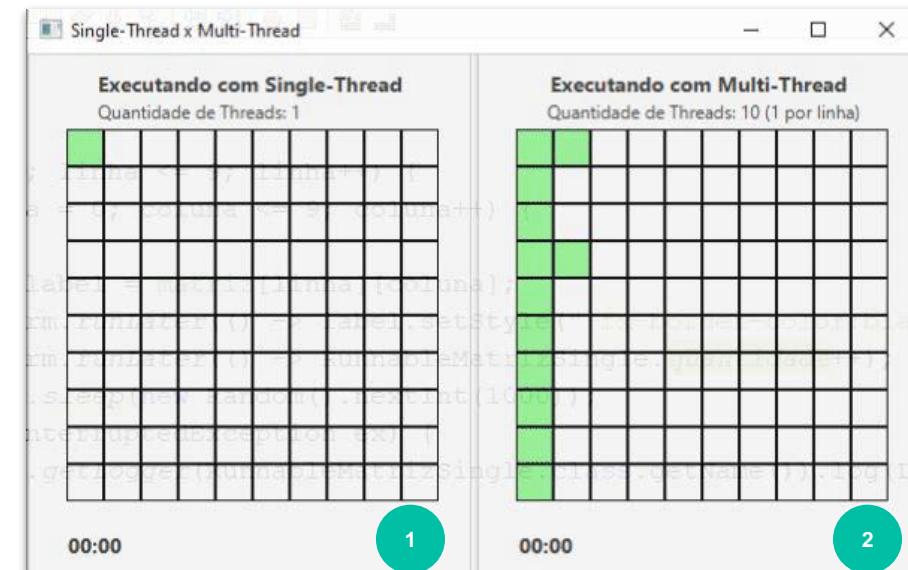
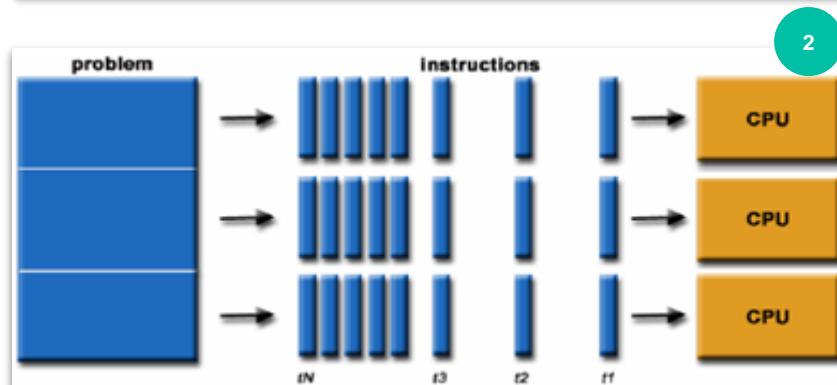
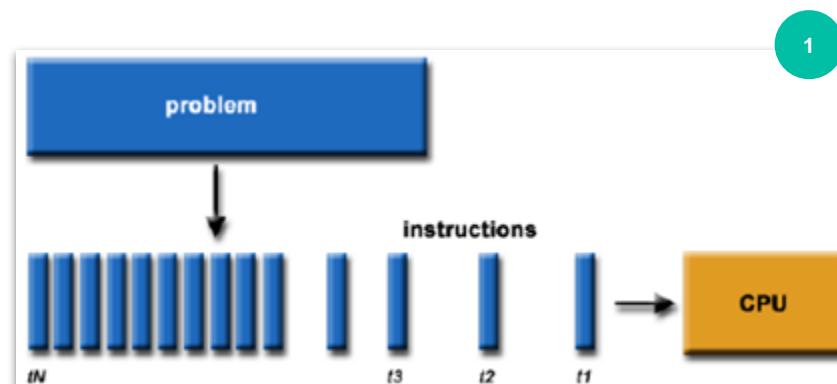
- 01** | Sequencial: uma Thread (Single-Thread)
- 02** | Concorrente: várias Threads (Multi-Thread)

Threads: linhas de execução separadas

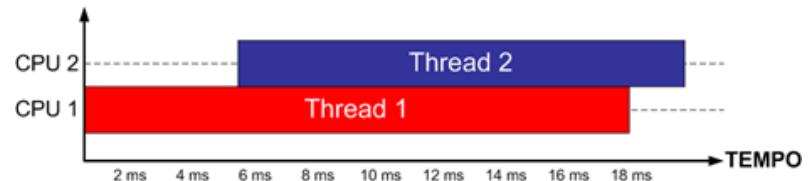
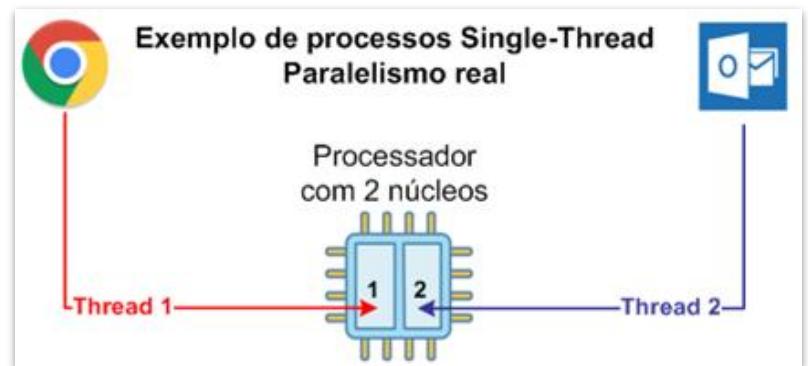
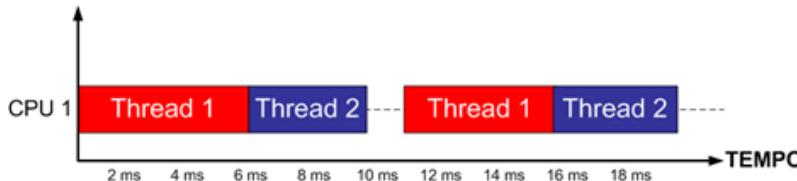
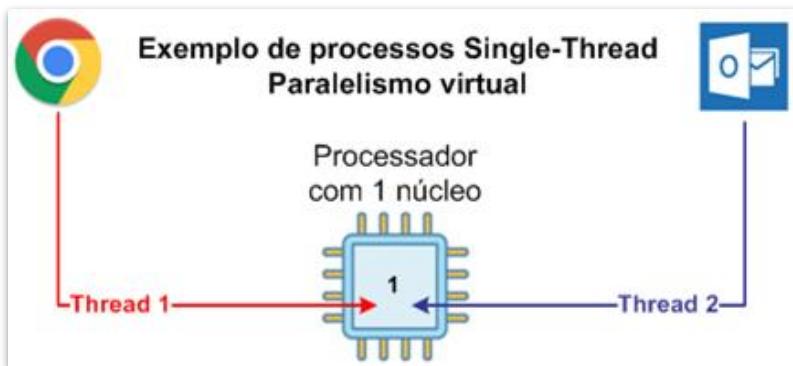


01 | Sequencial: uma Thread (Single-Thread)

02 | Concorrente: várias Threads (Multi-Thread)



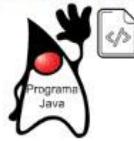
Modelo Single-Thread



Modelo Multi-Thread

 Single-Thread x Multi-Thread

Código de cada Thread



Programa Java

```

for(int i=1; i<1000; i++)
    if(i % 1 == 0)
        listView1.getItems().add(i);

for(int i=1; i<1000; i++)
    if(i % 2 == 0)
        listView2.getItems().add(i);

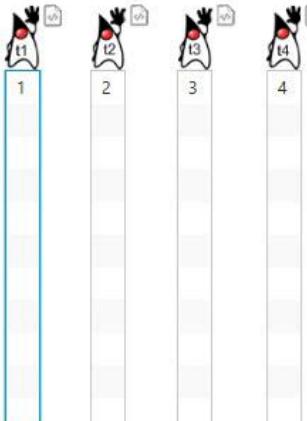
for(int i=1; i<1000; i++)
    if(i % 3 == 0)
        listView3.getItems().add(i);

for(int i=1; i<1000; i++)
    if(i % 4 == 0)
        listView4.getItems().add(i);

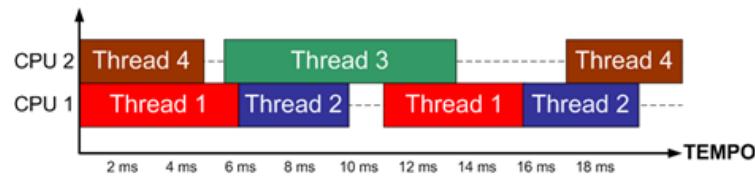
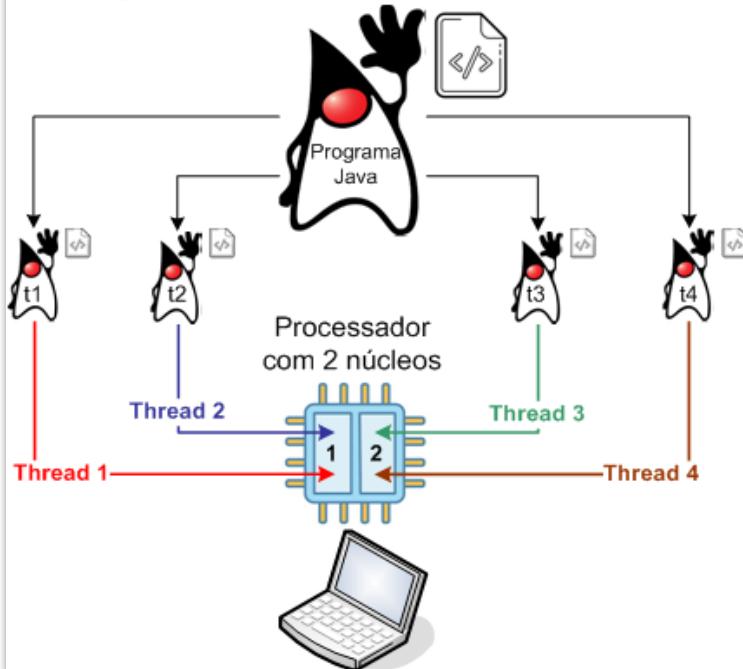
```

Executando com Multi-Thread

Quantidade de Threads: 4



Exemplo de processo Multi-Thread
Um programa em Java subdividido em Threads





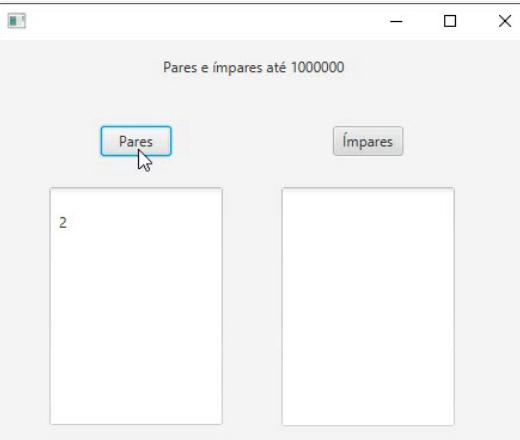
Introdução

Concorrência no Java: inclui primitivos de multithreading na própria linguagem e bibliotecas.

Exemplos de programação concorrente:

JAVA

Programa em Java para exibição de números pares e ímpares



JVM

Garbage
Collector

DOWNLOAD

Arquivo grande
Áudio ou Vídeo



2020/1 - SI - POO2: Video - Prog

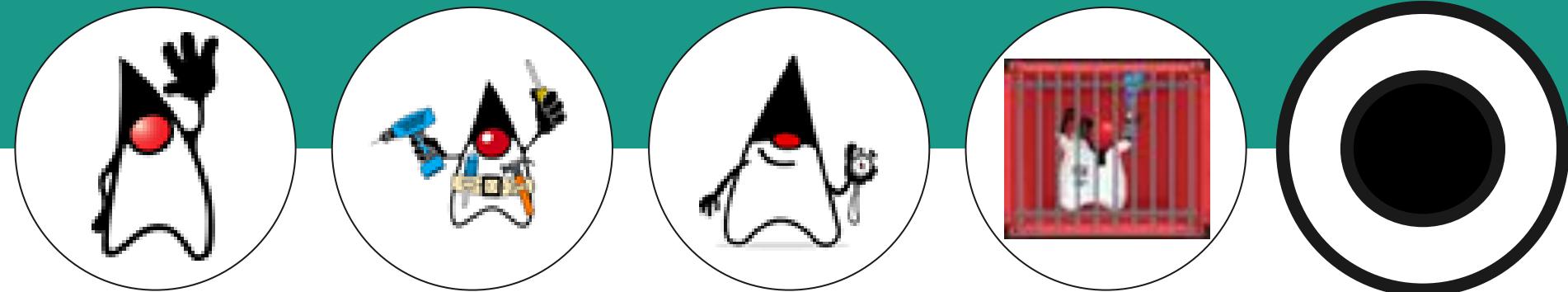
INSTITUTO FEDERAL
Espírito Santo | Centro de Referência em Formação
e em Desenvolvimento

Voltar para: Ca03: Programaç...

Sincronização de Thread
Problemas de Concorrência

- Por enquanto, os threads que vimos são **independentes**
- Em outras situações, threads devem **compartilhar dados** e são obrigados a se preocupar com o que os outros estão fazendo
- Um exemplo disso é a aplicação chamada "Produtor/Consumidor" em que um Produtor produz um fluxo de dados consumidos pelo consumidor
 - Exemplo 01: um thread (produtor) poderia estar gravando dados num arquivo enquanto outro thread (consumidor) lê dados deste arquivo;
 - Portanto, temos threads que **compartilham um recurso comum**
- Com tal compartilhamento, os threads devem se **sincronizar** para

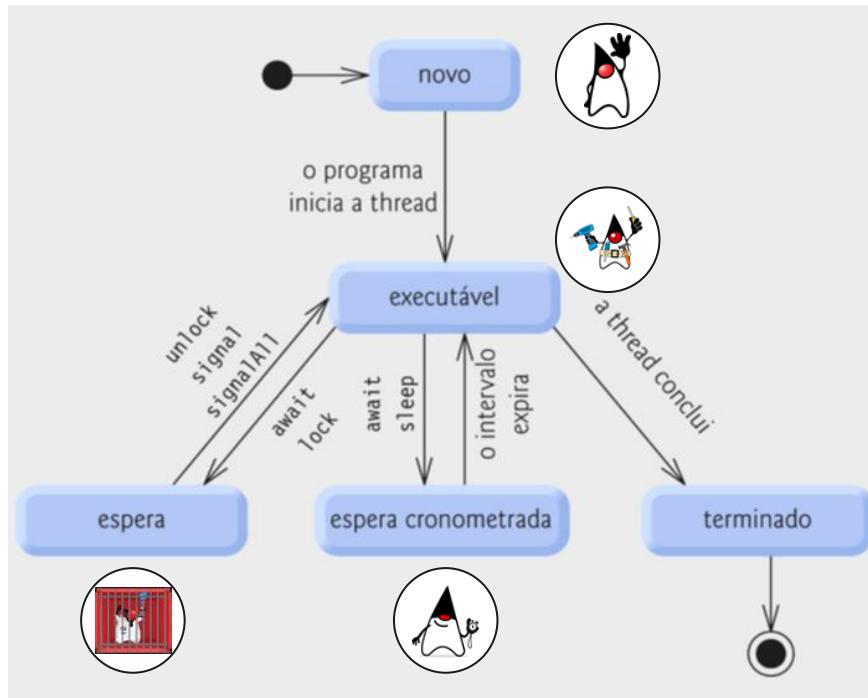
Programação Concorrente



Estados de Thread

Estados de thread

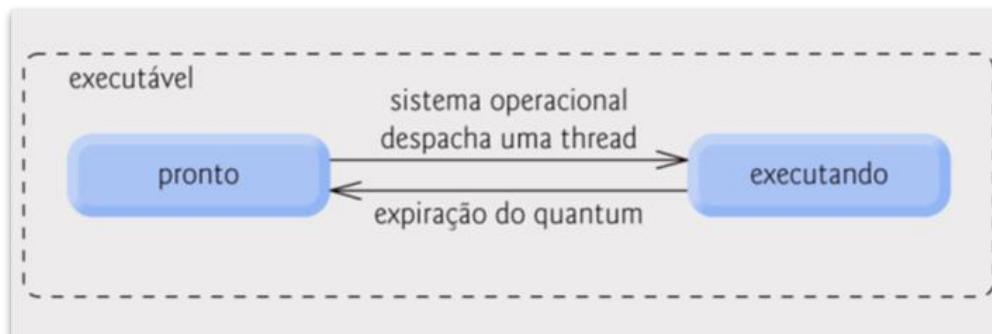
- uma nova thread inicia seu ciclo de vida no estado **novo**
- permanece nesse estado até o programa iniciar a thread, colocando-a no estado **executável**
- entra no estado de **espera** a fim de esperar que uma outra thread realize uma tarefa.
- entra em **espera cronometrada** para esperar uma outra thread ou para transcorrer um determinado período de tempo;
- uma thread executável transita para o estado **bloqueado** quando tenta realizar uma tarefa que não pode ser completada imediatamente e deve esperar temporariamente até que essa tarefa seja concluída;
- quando uma thread no estado executável completa sua tarefa ela entra no estado **terminado**.



Ciclo de vida de thread: visão do programador

Estados de thread

- **pronto:** uma thread nesse estado não está esperando uma outra thread, mas está esperando que o sistema operacional atribua a thread a um processador;
- **em execução:** uma thread nesse estado tem atualmente um processador e está executando. Uma thread no estado em execução frequentemente utiliza uma pequena quantidade de tempo de processador chamada fração de tempo, ou *quantum*, antes de migrar de volta para o estado pronto.



Ciclo de vida de thread: visão do sistema operacional

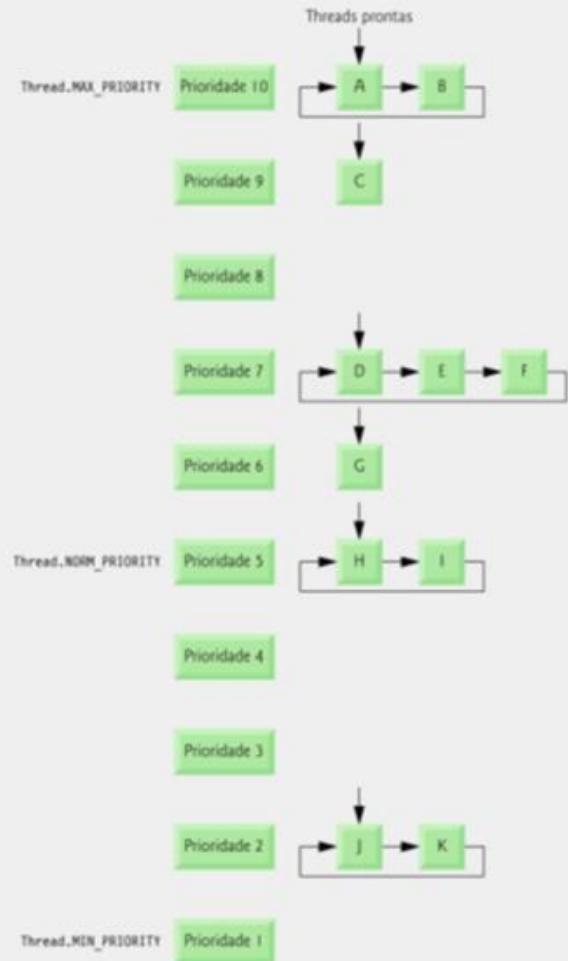
Programação Concorrente



Prioridade de Thread

Prioridades e Agendamentos

- Cada thread Java tem uma prioridade;
- As prioridades do Java estão no intervalo entre:
 - **MIN_PRIORITY** (uma constante de **1**) e
 - **MAX_PRIORITY** (uma constante de **10**);
- As threads com prioridade mais alta são mais importantes e terão um processador alocado antes das threads com prioridades mais baixas;
- A **prioridade padrão** é NORM_PRIORITY (constante de **5**).
- Em Java: **public void setPriority(xxx)**



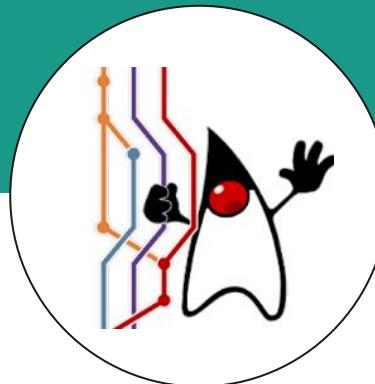
Prioridades e Agendamentos

Thread Scheduler: Agendador de Thread em um SO

- determina qual thread é executada em seguida;
- uma implementação simples executa threads com a mesma prioridade no estilo rodízio;
- threads de prioridade mais alta podem fazer **preempção** da thread atualmente **em execução**.
- em alguns casos, as threads de prioridade alta podem adiar indefinidamente threads de prioridade mais baixa; o que também é conhecido como **inanição**.



Programação Concorrente



Criando Threads

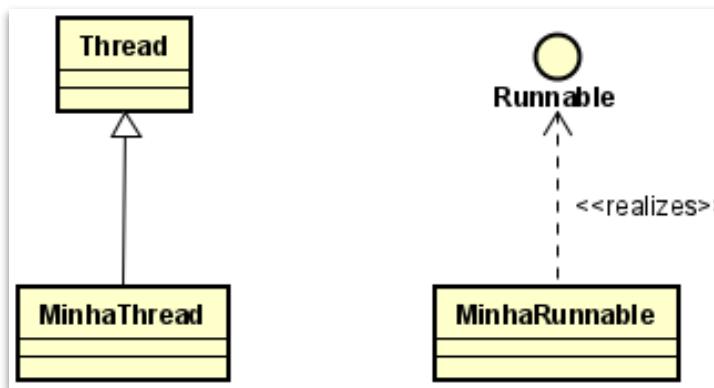


Criando e executando threads

Existem duas formas de criação de threads:

Extends Thread: herdar a classe Thread.

Implements Runnable: implementar a interface Runnable.

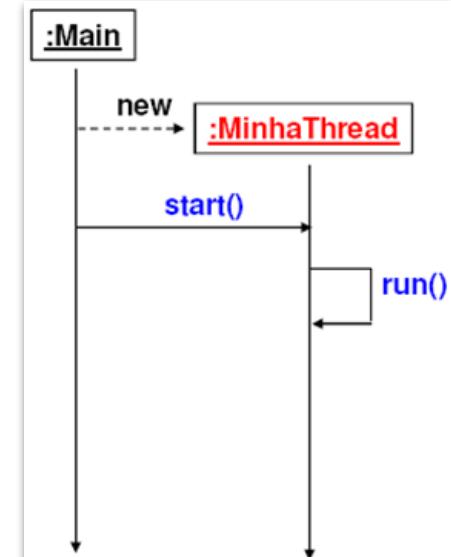


[threads-extends-thread-java](#)
[threads-implements-runnable-java](#)

Criando e executando threads

Criação de threads por **extends**

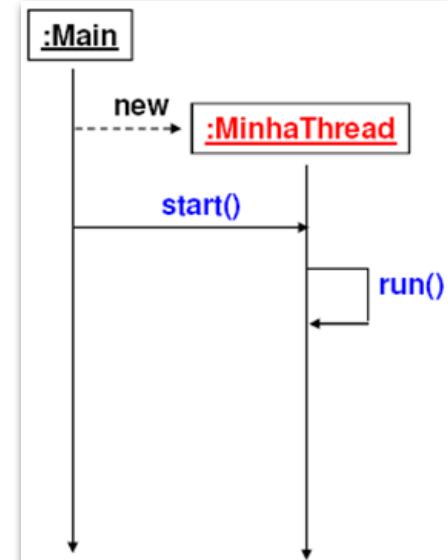
```
public class MinhaThread extends Thread{  
  
    public void run(){  
        System.out.println("Executando Thread!");  
    }  
}  
  
public class Main {  
  
    public static void main(String[] args) {  
        MinhaThread minhaThread = new MinhaThread();  
        minhaThread.start();  
    }  
}
```



Criando e executando threads

Criação de threads por **implements**

```
public class MinhaRunnable implements Runnable{  
    public void run(){  
        System.out.println("Executando Thread!");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        MinhaRunnable minhaRunnable = new MinhaRunnable();  
        Thread minhaThread = new Thread(minhaRunnable);  
        minhaThread.start();  
    }  
}
```



Criando threads

Exemplo básico de Threads contadoras:

- duas threads são criadas
- cada uma conta de 0 até 4
- os resultados podem ser intercalados

```
INICIO DA THREAD MAIN ***
thread 1 rodando
    thread 2 rodando
    thread 2: 0
    thread 2: 1
    thread 2: 2
thread 1: 0
thread 1: 1
    thread 2: 3
    thread 2: 4
    thread 2 FIM ***
thread 1: 2
thread 1: 3
thread 1: 4
thread 1 FIM ***
```

```
class MinhaRunnable implements Runnable {
    public void run() { //interface Runnable exige implementação de run
        String name = Thread.currentThread().getName();
        System.out.println(name + " rodando");
        for (int i=0;i < 5; i++) {
            System.out.println(name + ": " + i);
        }
        System.out.println(name + " FIM ***");
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("INICIO DA THREAD MAIN ***");
        Thread t1 = new Thread(new MinhaRunnable(), "thread 1");
        Thread t2 = new Thread(new MinhaRunnable(), "thread 2");
        t1.start();
        t2.start();
    }
}
```



Criando e executando threads

Métodos importantes

- **run()** - é o código que a thread executará.
- **start()** - sinaliza à JVM que a thread pode ser executada, mas saiba que essa execução não é garantida quando esse método é chamado, e isso pode depender da JVM.
- **isAlive()** - volta true se a thread está sendo executada e ainda não terminou.
- **sleep()** - suspende a execução da thread por um tempo determinado;
- **yield()** - torna o estado de uma thread executável para que thread com prioridades equivalentes possam ser processadas, isso será estudando mais adiante;
- **currentThread()** - é um método estático da classe Thread que volta qual a thread que está sendo executada.
- **getName()** - volta o nome da Thread, você pode especificar o nome de uma Thread com o método **setName()** ou na construção da mesma, pois existe os construtores sobrecarregados.



Criando e executando threads

Utilização dos seguintes métodos importantes:

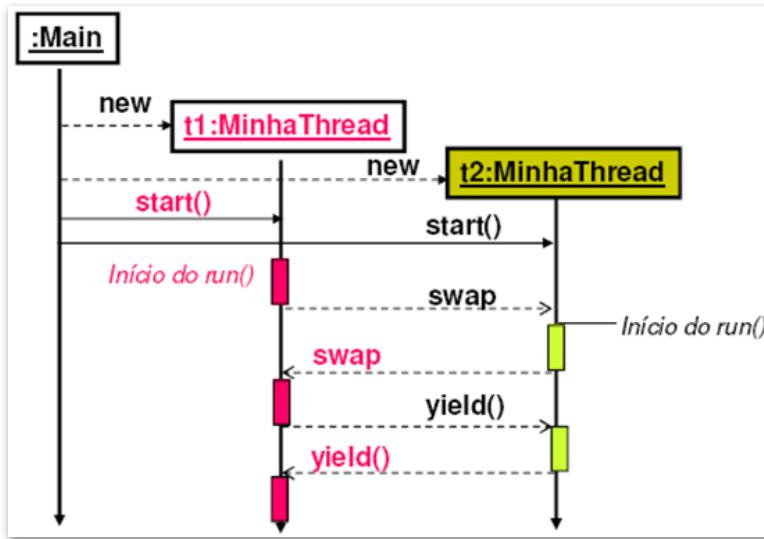
Yield

Sleep

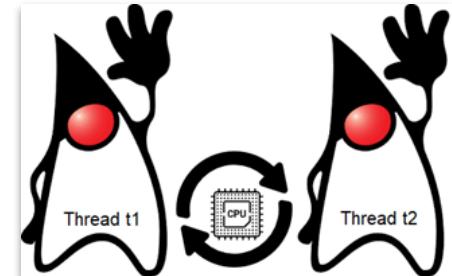
Join

Criando e executando threads

Yield: suspende a execução da thread atual e permite que outra ocupe o processador.



As flechas pontilhadas indicam swap;
O label das flechas indicam o motivo do swap;
Swap: escalonador do SO;
yield(): a chamada ao método causou a mudança.



Criando e executando threads: Yield

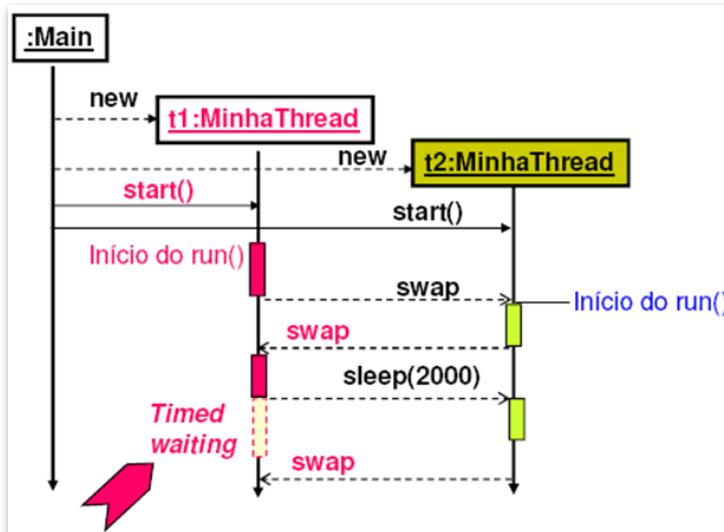
```
class MinhaRunnable implements Runnable {  
    public void run() {  
        String name = Thread.currentThread().getName();  
        System.out.println(name + " rodando");  
        for (int i=0;i < 5; i++) {  
            System.out.println(name + ": " + i);  
            // passando o controle para outra thread implicitamente  
            Thread.yield();  
        }  
        System.out.println(name + " FIM ***");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        System.out.println("INICIO DA THREAD MAIN ***");  
        Thread t1 = new Thread(new MinhaRunnable(), "thread 1");  
        Thread t2 = new Thread(new MinhaRunnable(), "\t\tthread 2");  
        t1.start();  
        t2.start();  
    }  
}
```

```
INICIO DA THREAD MAIN ***  
thread 1 rodando  
    thread 2 rodando  
thread 1: 0  
    thread 2: 0  
thread 1: 1  
    thread 2: 1  
thread 1: 2  
    thread 2: 2  
thread 1: 3  
    thread 2: 3  
thread 1: 4  
    thread 2: 4  
thread 1 FIM ***  
    thread 2 FIM ***
```

[threads-yield-java](#)

Criando e executando threads

Sleep: suspende a execução da thread atual por pelo menos x milisegundos e permite que outra ocupe o processador.



Criando e executando threads: Sleep

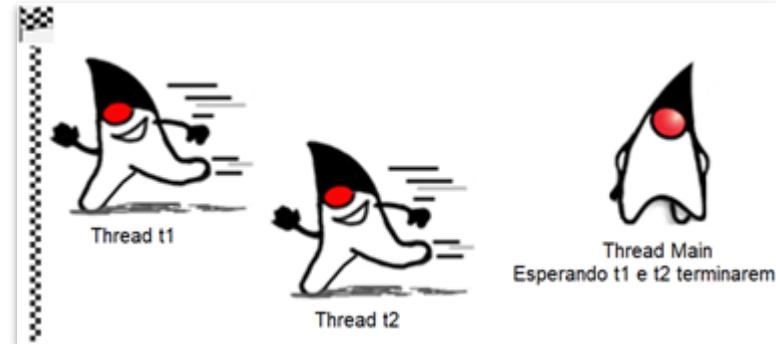
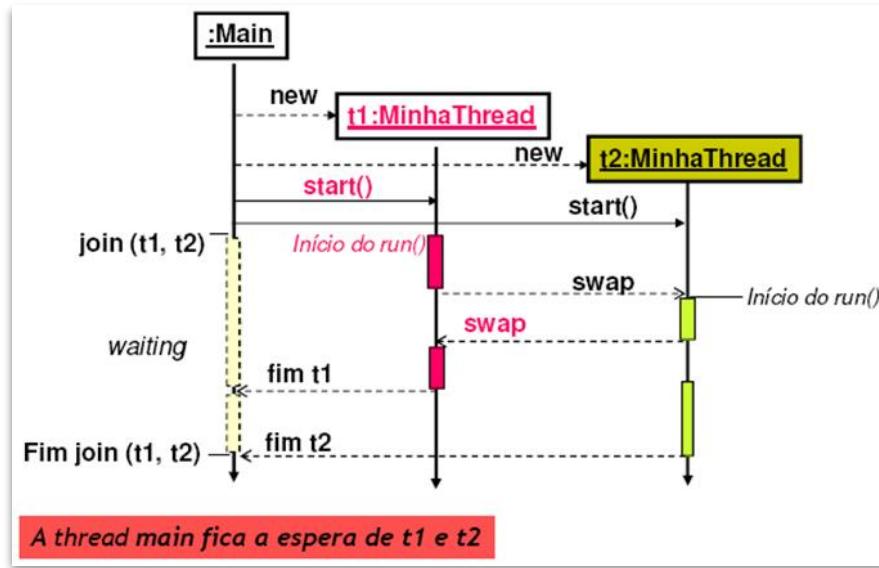
```
class MinhaRunnable implements Runnable {  
    public void run() {  
        String name = Thread.currentThread().getName();  
        System.out.println(name + " rodando");  
        for (int i = 0; i < 5; i++) {  
            System.out.println(name + ": " + i);  
            if (i % 3 == 0) {  
                Thread.sleep(2000); // dorme 2s a cada 3 contagens  
            }  
        }  
        System.out.println(name + " FIM ***");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        System.out.println("INICIO DA THREAD MAIN ***");  
        Thread t1 = new Thread(new MinhaRunnable(), "thread 1");  
        Thread t2 = new Thread(new MinhaRunnable(), "\t\tthread 2");  
        t1.start();  
        t2.start();  
    }  
}
```

```
INICIO DA THREAD MAIN ***  
thread 1 rodando  
thread 1: 0  
                    thread 2 rodando  
                    thread 2: 0  
thread 1: 1  
thread 1: 2  
thread 1: 3  
                    thread 2: 1  
                    thread 2: 2  
                    thread 2: 3  
thread 1: 4  
thread 1 FIM ***  
                    thread 2: 4  
                    thread 2 FIM ***
```

[threads-sleep-java](#)

Criando e executando threads

Join: permite que uma thread espere pelo término de duas ou mais threads.



Criando e executando threads: Join

```
class MinhaRunnable implements Runnable {  
    public void run() {  
        String name = Thread.currentThread().getName();  
        System.out.println(name + " rodando");  
        for (int i=0;i < 5; i++)  
            System.out.println(name + ": " + i);  
        System.out.println(name + " FIM ***");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        System.out.println("INICIO DA THREAD MAIN ***");  
        Thread t1 = new Thread(new MinhaRunnable(), "thread 1");  
        Thread t2 = new Thread(new MinhaRunnable(), "\t\tthread 2");  
        t1.start();  
        t2.start();  
        try {  
            t1.join(); // a thread main aguarda o término de t1  
            t2.join(); // a thread main aguarda o término de t2  
        }  
        System.out.println("As duas threads encerraram a contagem");  
    }  
}
```

```
INICIO DA THREAD MAIN ***  
thread 1 rodando  
thread 1: 0  
thread 1: 1  
thread 1: 2  
thread 1: 3  
thread 1: 4  
thread 1 FIM ***  
                                thread 2 rodando  
                                thread 2: 0  
                                thread 2: 1  
                                thread 2: 2  
                                thread 2: 3  
                                thread 2: 4  
                                thread 2 FIM ***  
*** As duas threads encerraram a contagem * * *
```

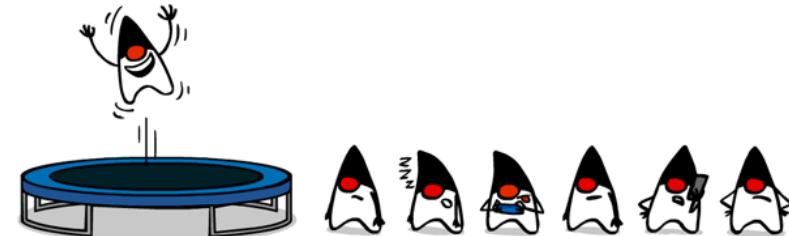
Programação Concorrente



Sincronização de Thread



Sincronização de Thread



Problemas de concorrência

- Por enquanto, os threads que vimos são independentes
- Em outras situações, threads devem compartilhar dados e são obrigados a se preocupar com o que os outros estão fazendo
- Um exemplo disso é a aplicação chamada "**Produtor/Consumidor**" em que um Produtor produz um fluxo de dados consumidos pelo consumidor
 - **Exemplo 01:** um thread (produtor) poderia estar gravando dados num arquivo enquanto outro thread (consumidor) lê dados deste arquivo;
 - Portanto, temos threads que compartilham um recurso comum
- Com tal compartilhamento, os threads devem se **sincronizar** para acessar o recurso

Sincronização de Thread

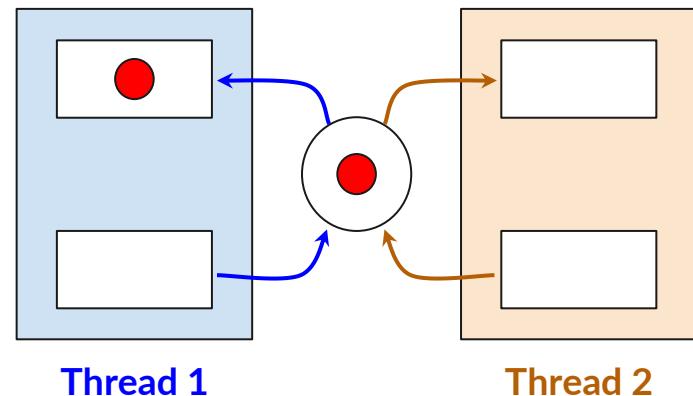
Problemas de concorrência

- Processos podem compartilhar dados
- Sincronizar acesso aos dados é necessário
- **Região / Seção crítica:** necessita de atomicidade
- Para garantir atomicidade, **exclusão mútua**

Exemplo

- x é uma variável compartilhada ($x=0$)
- Thread 1 faz: $x = x + 1$
- Thread 2 faz: $x = x + 1$
- x deve ser 2 ao final
- $x = x + 1$ deve ser executada **atomicamente**

processos querem o recurso, mas só um pode utilizá-lo, caso contrário o recurso pode ficar num estado inconsistente

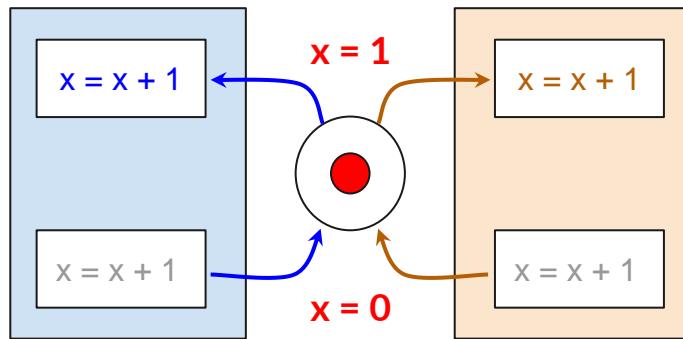


Quem ganha a disputa?



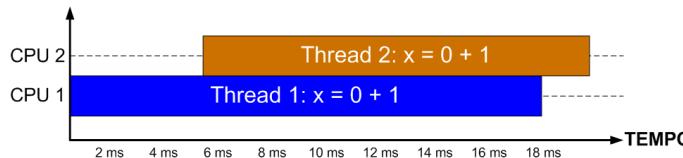
Sincronização de Thread

Sem exclusão mútua

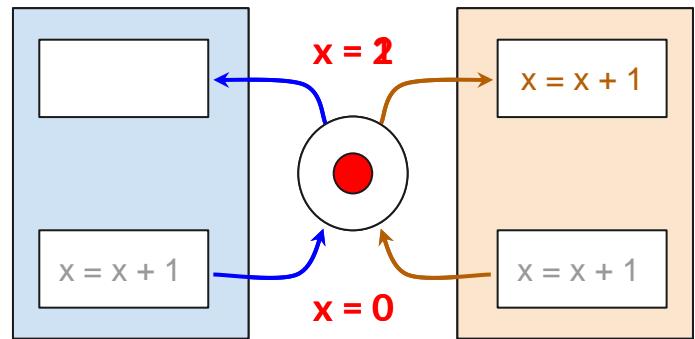


Thread 1

Thread 2

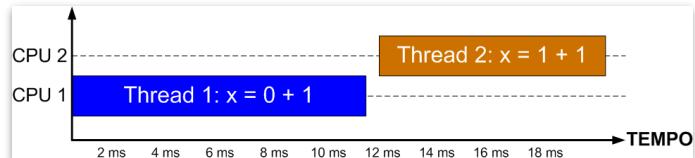


Com exclusão mútua



Thread 1

Thread 2



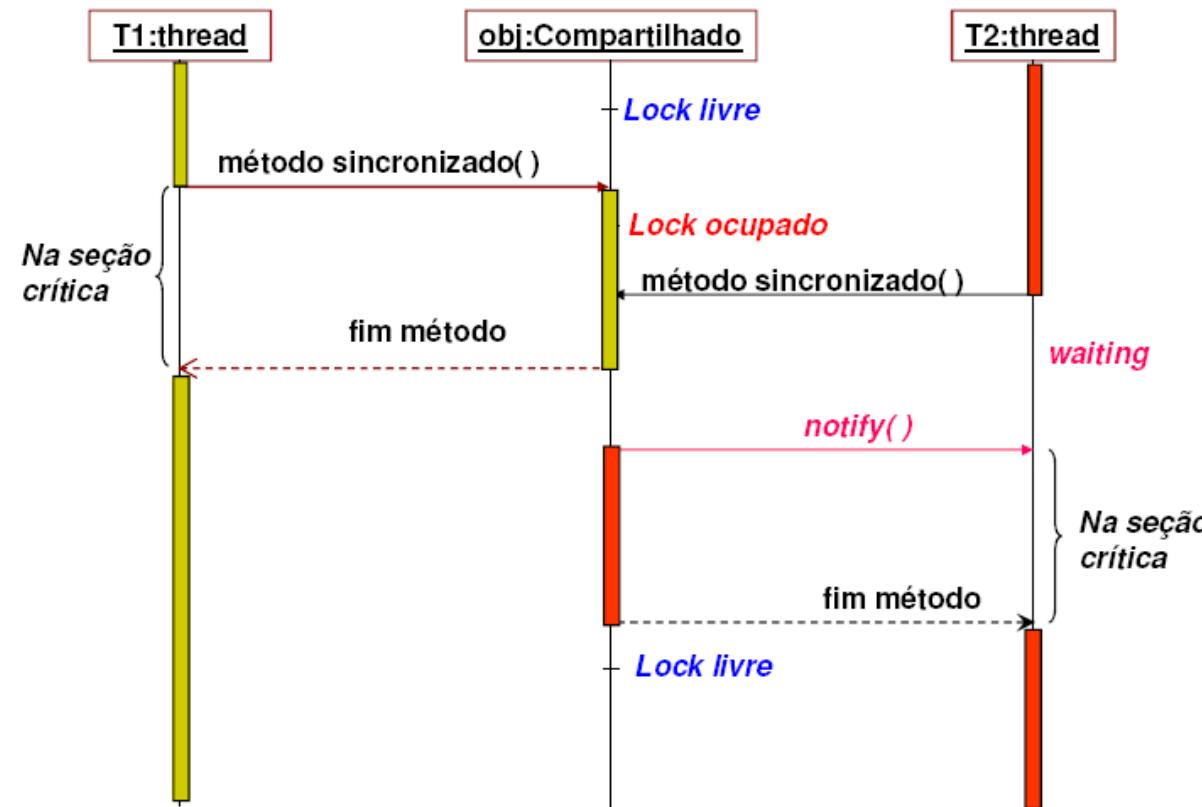


Sincronização de Thread

Problemas de concorrência

- Como resolver os problemas de acesso à seção crítica?
- **Monitor**: mais alto nível e mais fácil de utilizar
- Java só trabalha com monitores
- Todo objeto em Java é um monitor
- Quando uma thread executa um **bloco de código sincronizado**, ela fecha o acesso às demais. Outra thread que tentar acessar a seção crítica (o bloco de código) não conseguirá até que a primeira libere o acesso.
- Em java, o **bloqueio é feito no objeto** e não no método.
- Útil quando duas ou mais threads devem atualizar a mesma área de memória (atributo compartilhado).

EXEMPLO MONITOR LOCK



Sincronização de Thread

Problemas de concorrência

- **A 1^a sincronização: synchronized**
 - Dentro de um programa, segmentos de código que acessam os mesmos dados usando threads diferentes e concorrentes são chamados de regiões críticas ou seções críticas
 - Uma seção crítica pode ser um bloco de statements ou um método inteiro e deve ser identificada com a palavra synchronized
 - A JVM fornece um lock para cada objeto e o objeto é travado (o lock é "obtido") ao entrar numa seção crítica
- **A 2^a sincronização: notify, notifyAll e wait**

SINTAXE PARA SYNCHRONIZED

São construções equivalentes

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

```
public void metodo() {  
    synchronized (this) {  
        ...  
        ...  
        ...  
    }  
}
```

Métodos estáticos podem ser sincronizados - equivale a um lock de classe

Sincronização de Thread

Synchronized



[threads-synchronized1iphone-java](#)
[threads-synchronized2iphone-java](#)

- Cenário: Imaginemos um **processo de compra pela Internet**, onde inúmeras pessoas podem consultar os itens disponíveis em estoque e realizar seus pedidos. Pois bem, como não queremos causar situações indigestas com nossos clientes, precisamos garantir que seus pedidos sejam faturados corretamente. Bom onde queremos chegar?
- Imagine que **temos 5 aparelhos iPhone 11** em nosso estoque e que foi lançado uma promoção desse aparelho. Temos ainda que **200 pessoas** estão dispostas a entrar no tapa por um aparelho
- Bem, temos que garantir que esse processo seja concretizado sem maiores problemas... Vejamos como resolver esse problema:

Sincronização de Thread: Synchronized



```
class Produto implements Runnable {  
    private int estoque = 5;  
  
    public void run() {  
        efetuarPedido();  
    }  
  
    public void efetuarPedido() {  
        try {  
            if (this.estoque > 0) {  
                System.out.println("Pedido faturado para o cliente "+Thread.currentThread().getName());  
                Thread.sleep(250);  
                this.estoque--;  
            } else {  
                System.out.println("Não tem estoque para o cliente "+Thread.currentThread().getName());  
            }  
        } catch (Exception ex) {  
        }  
    }  
}
```

```
public class PedidoCompra {  
    public static void main(String[] args) {  
        Produto p = new Produto(5);  
        Thread[] t = new Thread[15];  
        for (int i=0; i < t.length; i++ ) {  
            t[i] = new Thread(p);  
            t[i].setName("Cliente: "+i);  
            t[i].start();  
        }  
    }  
}
```

Não tente vender o programa acima!

O código sempre efetuará o pedido tendo ou não estoque.

Sincronização de Thread: Synchronized



```
class Produto implements Runnable {  
    private int estoque = 5;  
  
    public void run() {  
        efetuarPedido();  
    }  
  
    public synchronized void efetuarPedido() {  
        try {  
            if (this.estoque > 0) {  
                System.out.println("Pedido faturado para o cliente "+Thread.currentThread().getName());  
                Thread.sleep(250);  
                this.estoque--;  
            } else {  
                System.out.println("Não tem estoque para o cliente "+Thread.currentThread().getName());  
            }  
        } catch (Exception ex) {  
        }  
    }  
}  
  
public class PedidoCompra {  
    public static void main(String[] args) {  
        Produto p = new Produto(5);  
        Thread[] t = new Thread[15];  
        for (int i=0; i < t.length; i++ ) {  
            t[i] = new Thread(p);  
            t[i].setName("Cliente: "+i);  
            t[i].start();  
        }  
    }  
}
```

Modificador synchronized

Não deixa 2 threads executarem o método ao mesmo tempo



Sincronização de Thread

wait notify notifyAll

- Interação entre Segmentos
 - wait()
 - notify()
 - notifyAll()
- O método **Object.wait()** interrompe a thread atual, ou seja, coloca a mesma para “dormir” até que uma outra thread use o método **Object.notify()** no mesmo objeto para “acordá-la”.

MONITOR LOCK: WAIT, NOTIFY

- ◊ Cada monitor tem uma fila de processos bloqueados
- ◊ 3 métodos especiais podem ser usados dentro do monitor

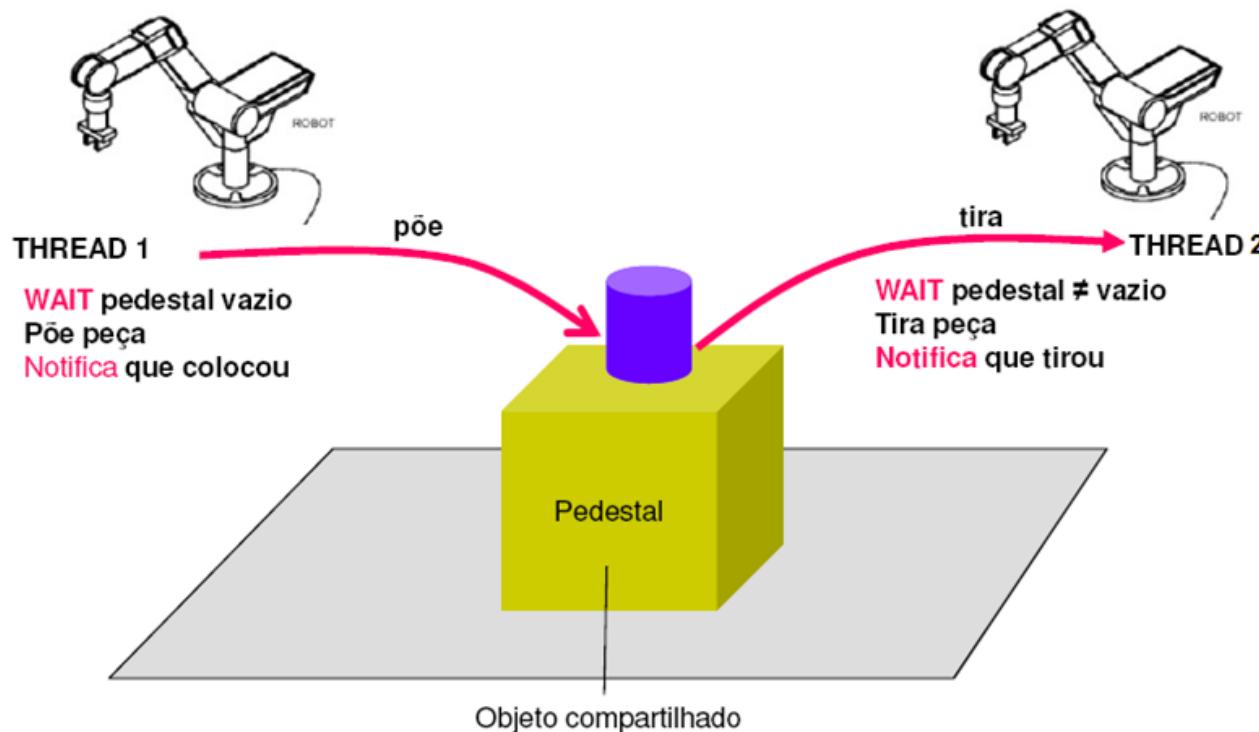
```
synchronized  
    (object) {  
    ...  
    object.wait();  
    ...  
    object.notify();  
    ...  
    object.notifyAll(  
    );  
    ...  
}
```

Thread Libera o lock do *object* e
passa ao estado *waiting*

Se a fila não estiver vazia, pega **um**
processo qualquer da fila e o
desbloqueia

Se a fila não estiver vazia,
desbloqueia todos os processos da
fila que vão disputar o lock - mas só
um será escolhido pelo escalonador
(ineficiente se houver muitas threads)

MONITOR LOCK: WAIT/NOTIFY



MONITOR LOCK: WAIT/NOTIFY

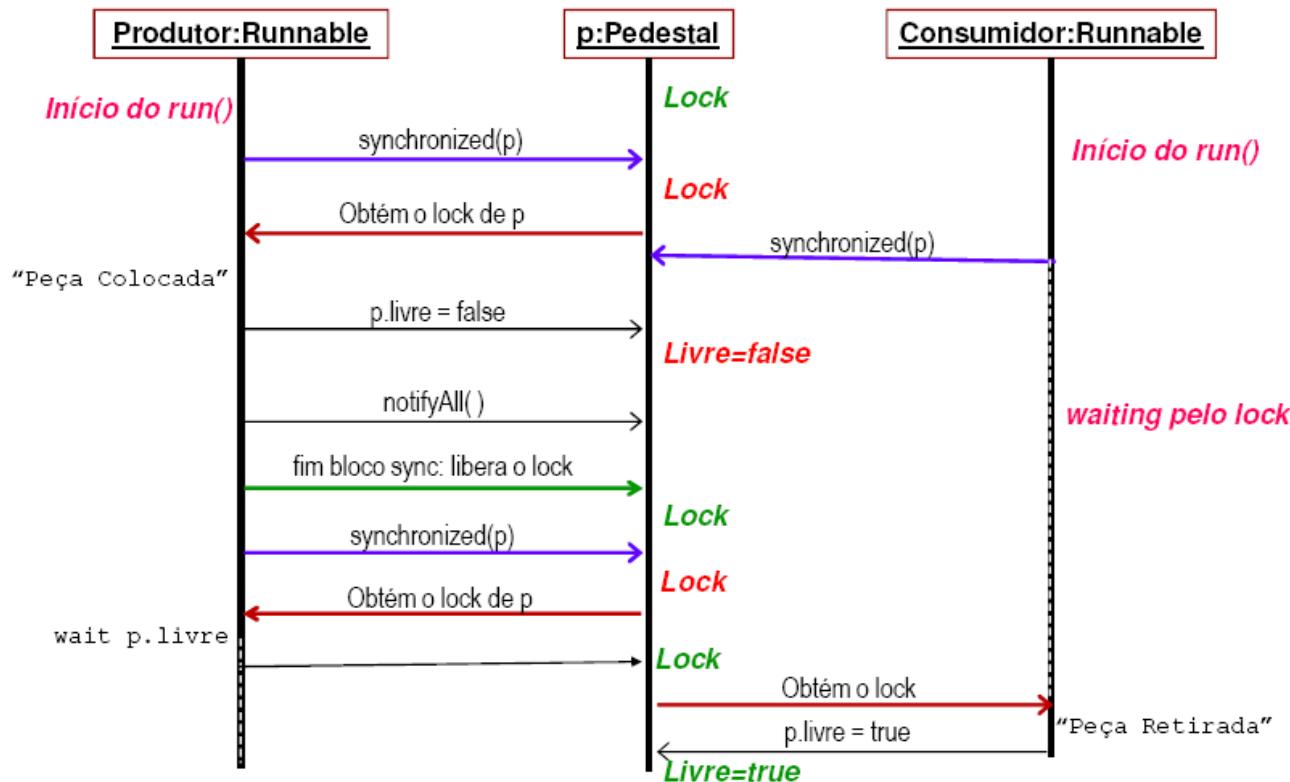
```
class Pedestal {  
    boolean livre = true;  
}
```

```
class RoboProdutor implements Runnable {  
    Pedestal p;  
    RoboProdutor(Pedestal p) {  
        this.p = p;  
    }  
    public void run() {  
        while (true) {  
            synchronized (p) {  
                while (!p.livre) {  
                    p.wait();  
                }  
                println("Peça colocada");  
                p.livre=false;  
                p.notifyAll();  
            }  
        }  
    }  
}
```

```
class RoboConsumidor implements Runnable {  
    Pedestal p;  
    RoboConsumidor(Pedestal p) {  
        this.p = p;  
    }  
    public void run() {  
        while (true) {  
            synchronized (p) {  
                while (p.livre) {  
                    p.wait();  
                }  
                println("Peça retirada");  
                p.livre=true;  
                p.notifyAll();  
            }  
        }  
    }  
}
```

Catch e try foram omitidos
Wait libera o lock; Notify não libera o lock

EXEMPLO MONITOR LOCK



Sincronização de Thread

Join x Wait

- Quando utilizar Join ou Wait ?

A começar pelo **join**, este espera até que a thread seja totalmente finalizada, ou seja, seu processamento termine.

Diferentemente do **wait** que já libera a thread após o notify, e não necessariamente a thread que chamou o notify precisa ter terminado.

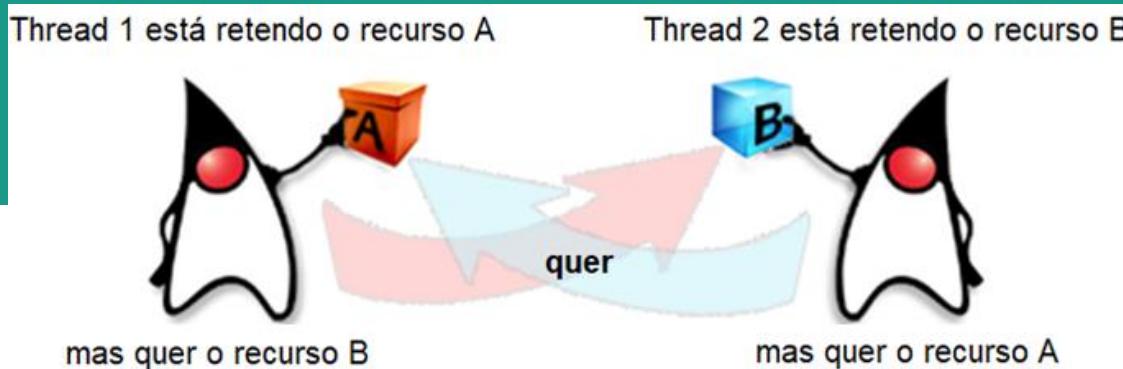
```
//Usando Join  
synchronized(two){  
    two.join()  
}
```

```
//Usando Wait  
synchronized(two){  
    two.wait()  
}
```

....

```
synchronized(two){  
    notify();  
    //or notifyAll();  
}
```

Programação Concorrente

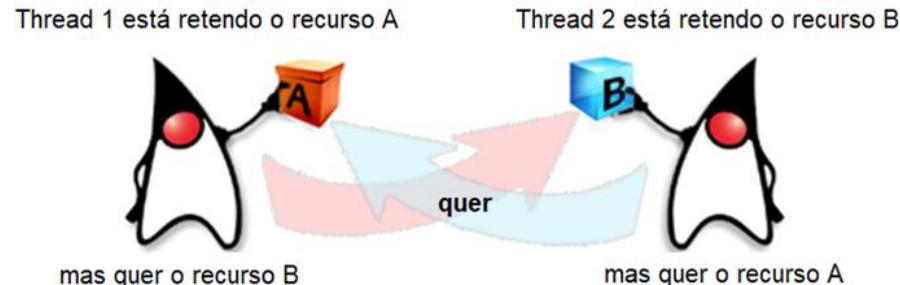


Deadlock

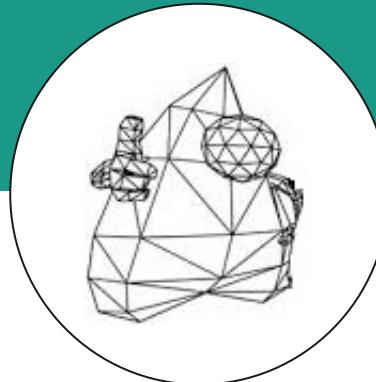
Sincronização de Thread

Deadlock

- O **impasse** (deadlock) ocorre quando uma thread em espera (vamos chamá-la de thread1) não pode prosseguir porque está esperando (direta ou indiretamente) outra thread (vamos chamá-la de thread2) prosseguir;
- Simultaneamente, a thread2 não pode prosseguir porque está esperando (direta ou indiretamente) a thread1 prosseguir.
- Como duas threads estão esperando uma à outra, as ações que permitiriam a cada thread continuar **a execução nunca ocorre**.



Programação Concorrente



Multithreading com GUI



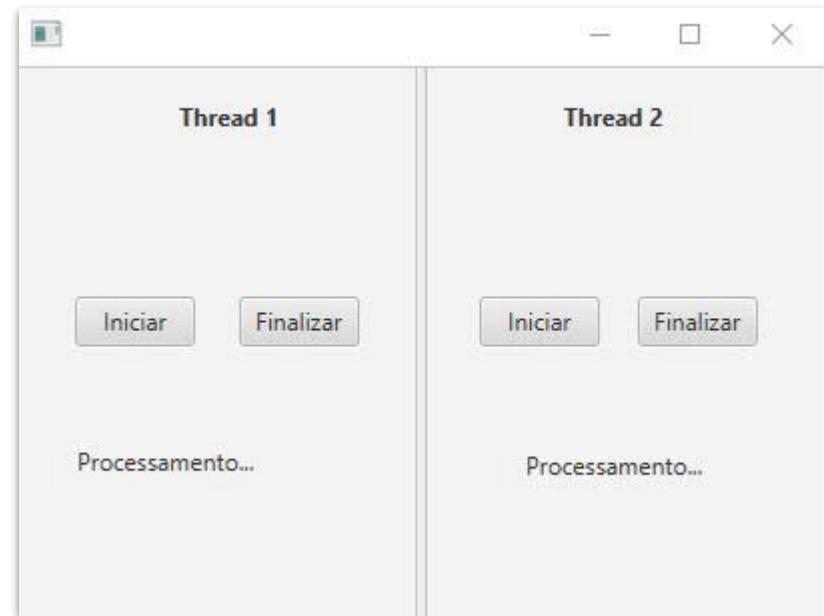
Multithreading com GUI



- JavaFX, como Swing e AWT, usa uma única Thread para processar todos os eventos da interface do usuário, chamada **JavaFX Application Thread**.
- Todas as tarefas que exigem interação com a GUI de um aplicativo são colocadas em uma fila de eventos e executados em sequência pela thread.
- Os componentes GUI JavaFX não são seguros para threads – não podem ser manipulados por múltiplas threads sem o risco de resultados incorretos.

Multithreading com GUI

Exemplo de Thread com JavaFX





```
public class MinhaRunnable implements Runnable {

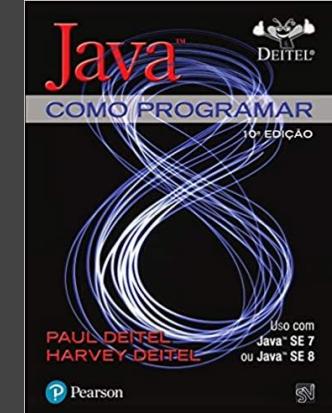
    Label label;
    int i;

    public MinhaRunnable(Label j) {
        label = j;
    }

    @Override
    public void run() {
        for (i = 1; i <= 5; i++) {
            Platform.runLater(() -> label.setText("Processando..." + i));
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                Logger.getLogger(MinhaRunnable.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
        Platform.runLater(() -> label.setText("Finalizada!"));
    }
}
```

Referências Bibliográficas

H. M. Deitel, P. J. Deitel. Java: Como Programar, **Capítulo 23 – Concorrência**, 10^a Edição. Pearson, 2016.





Obrigado.



Sobre mim



Rafael Mesquita, Prof.

Prof. Dr. Formado em
Ciência da Computação
pela Universidade Federal
de Lavras