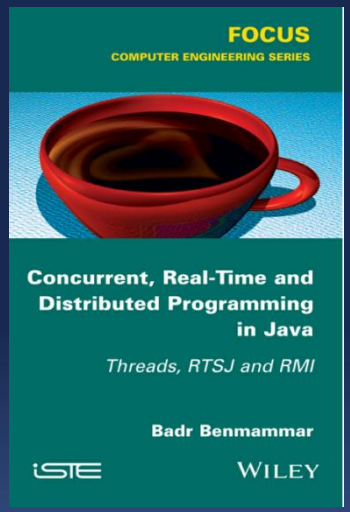
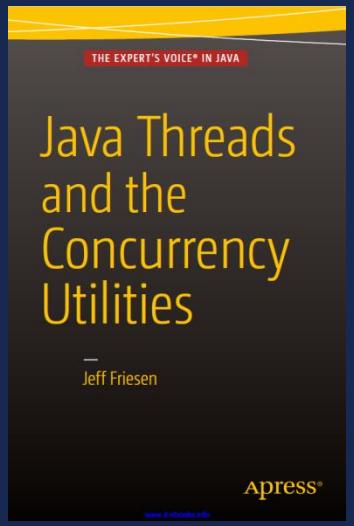
Programação Paralela e Concorrente

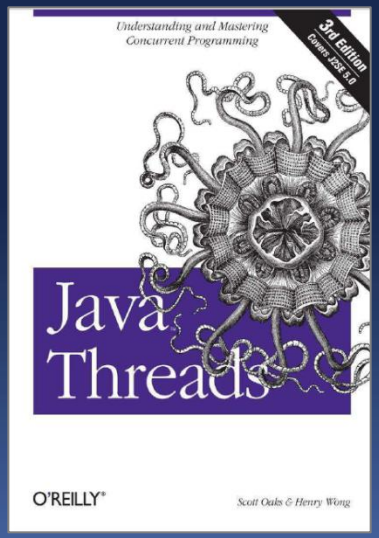
Unidade 6 – Sincronização de Threads

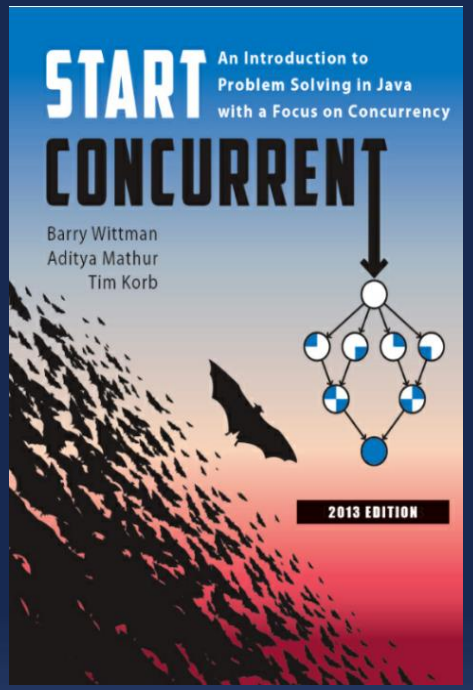
Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecidovfreitas@gmail.com

Bibliografia









Introdução

- Até o presente momento implementamos algumas aplicações que empregaram threads de **forma independente**;
- Ou seja, os **dados** manipulados por um thread não eram compartilhados com outros threads;
- Estudaremos agora a situação no qual um determinado recurso é **compartilhado** entre diversos threads;
- Esse **compartilhamento** de recursos comumente ocorre ao se empregar **Programação Concorrente**.

O que ocorre quando dois ou mais threads acessam um mesmo objeto modificando seu estado ?

Threads compartilhando Recursos

- Para responder a essa questão, escreveremos uma aplicação que irá simular um **Banco** que possui diversos clientes com contas bancárias;
- Simularemos nessa aplicação **transações** que movimentam dinheiro entre as diversas contas correntes;

Classe Banco

- Vamos considerar que o Banco a ser empregado na aplicação possue 500 contas correntes;
- Para efeito de definição do estado inicial da aplicação, consideraremos que todas as contas correntes do Banco possuem um **saldo inicial** de R$ 1000,00 cada uma.
- Assim, nesse estado inicial, o saldo total do banco (considerando todas as contas correntes) será de 500 x R$ 1000,00 = R$ 500.000,00.

Classe Banco

- Na classe **Banco** iremos definir um array de **500** contas com o saldo de cada corrente, inicialmente com o valor de **R$ 1000,00** em cada conta;
- Definiremos também um atributo referente ao **valor máximo** de **transferência** entre contas igual a **R$ 1000,00**.

```
package br.uscs;

public class Banco {

    private final double[] tabContas;
    private final static double maxTransfer = 1000.00;

    public Banco() {

        this.tabContas = new double[500];

        for (int i = 0; i < tabContas.length; i++) {
            tabContas[i] = 1000.0;
        }
    }
```

Classe Banco

- Definição de getters e setters;

```
public static double getMaxtransfer() {
    return maxTransfer;
}

public double[] getTabContas() {
    return tabContas;
}

public int size() {
    return tabContas.length;
}
```

Classe Banco

- Definição da função **getSaldoTotalBanco()** que irá **retornar** o valor total armazenado em todas as contas do Banco.

```
    public double getSaldoTotalBanco() {
        double saldoTotalBanco = 0;
        for (int i = 0; i < tabContas.length; i++ )
            saldoTotalBanco = saldoTotalBanco + tabContas[i];
        return saldoTotalBanco;
    }
}
```

Classe Transacao

- Vamos agora definir uma classe que irá **abstrair** transações de movimentações de valores de uma conta para outra conta do Banco;
- Essa classe acessará o array de contas do Banco e, assim, definirmos o objeto **Banco** como atributo da classe **Transacao**;
- Simularemos na função **movimento()** a conta de **origem**, a conta de **destino** e o **valor** a ser transferido de uma conta para outra.

```
package br.uscs;

public class Transacao {
    private Banco b;

    public Transacao(Banco b) {
        this.b = b;
    }

    public void movimento() {

        int contaOrigem = (int) (b.size() * Math.random());
        int contaDestino = (int) (b.size() * Math.random());

        double valorTransferencia = Banco.getMaxtransfer() * Math.random();
        this.transfer(contaOrigem, contaDestino, valorTransferencia);
    }
```

Classe Transacao

- Definiremos nessa classe, a função **transfer()** que irá processar a tarefa de **transferência** de valores de uma conta para outra.

```
public void transfer(int contaOrigem ,int contaDestino,double valorTransferencia) {

    System.out.printf("~~~~~~ Saldo anterior da conta %d: %f \n",
                      contaOrigem, b.getTabContas()[contaOrigem]);
    System.out.printf("~~~~~~ Saldo anterior da conta %d: %f \n",
                      contaDestino, b.getTabContas()[contaDestino]);

    if (b.getTabContas()[contaOrigem] < valorTransferencia )
        System.out.println("**** Saldo Insuficiente ****\n");
    else {
        b.getTabContas()[contaOrigem] = b.getTabContas()[contaOrigem] - valorTransferencia;
        b.getTabContas()[contaDestino] = b.getTabContas()[contaDestino] + valorTransferencia;

        System.out.printf("\n=============> %10.2f transferido da conta:  %d para a conta: %d\n\n",
                valorTransferencia, contaOrigem, contaDestino);

        System.out.printf("~~~~~~ Saldo posterior da conta %d: %f \n",
                contaOrigem, b.getTabContas()[contaOrigem]);
        System.out.printf("~~~~~~ Saldo posterior da conta %d: %f \n\n\n",
                contaDestino, b.getTabContas()[contaDestino]);
    }
  }
}
```

Processando Transação

- Inicialmente, **inicializaremos** um objeto da classe Banco;
- Considerando que inicialmente há **500** contas com valor de **R$ 1.000,00** em cada uma, o valor total armazenado será de **R$ 500.000,00**.

```
package br.uscs;

import java.text.DecimalFormat;

public class TesteTransfer {

    public static void main(String[] args) {

        DecimalFormat df = new DecimalFormat();
        df.applyPattern("R$ #0.00");

        Banco b = new Banco();

        System.out.println("\n**** Saldo Total do Banco antes das Transações : "
                                    + df.format(b.getSaldoTotalBanco()) + "\n\n");
```

Processando Transação

- Vamos agora processar **10 transações** que irão processar de forma **sequencial**, fazendo movimentações de valores aleatórios em contas também aleatórias do Banco, acessando dessa forma o **objeto tabContas** da classe Banco.
- Ao **final** do processamento, **o saldo total do banco** deverá ser o **mesmo** que o **saldo total inicial** (R$ 500.000,00)

```
        for (int i = 0 ; i<10; i++) {
            Transacao t = new Transacao(b);
            System.out.println("$$$$$$$Processando Transação T"+ i + " $$$$$$$$");
            t.movimento();
        }

        Transacao t = new Transacao(b);
        t.movimento();

        System.out.println("\n**** Saldo Total do Banco após as Transações: "
                                    + df.format(b.getSaldoTotalBanco()) + "\n\n");

    }
}
```

Log de Execução

```
**** Saldo Total do Banco antes das Transações : R$ 500000.00


$$$$$$$Processando Transação T0 $$$$$$$$
~~~~~~ Saldo anterior da conta 355: 1000.000000
~~~~~~ Saldo anterior da conta 477: 1000.000000

==============>     409.68 transferido da conta:  355 para a conta: 477

~~~~~~ Saldo posterior da conta 355: 590.323913
~~~~~~ Saldo posterior da conta 477: 1409.676087


$$$$$$$Processando Transação T1 $$$$$$$$
~~~~~~ Saldo anterior da conta 60: 1000.000000
~~~~~~ Saldo anterior da conta 168: 1000.000000

==============>     873.87 transferido da conta:  60 para a conta: 168

~~~~~~ Saldo posterior da conta 60: 126.134629
~~~~~~ Saldo posterior da conta 168: 1873.865371
```

Log de Execução

```
$$$$$$$Processando Transação T2 $$$$$$$$
~~~~~~ Saldo anterior da conta 466: 1000.000000
~~~~~~ Saldo anterior da conta 203: 1000.000000

==============>     119.33 transferido da conta:  466 para a conta: 203

~~~~~~ Saldo posterior da conta 466: 880.673807
~~~~~~ Saldo posterior da conta 203: 1119.326193


$$$$$$$Processando Transação T3 $$$$$$$$
~~~~~~ Saldo anterior da conta 186: 1000.000000
~~~~~~ Saldo anterior da conta 201: 1000.000000

==============>     169.72 transferido da conta:  186 para a conta: 201

~~~~~~ Saldo posterior da conta 186: 830.277195
~~~~~~ Saldo posterior da conta 201: 1169.722805


$$$$$$$Processando Transação T4 $$$$$$$$
~~~~~~ Saldo anterior da conta 430: 1000.000000
~~~~~~ Saldo anterior da conta 238: 1000.000000

==============>     965.72 transferido da conta:  430 para a conta: 238

~~~~~~ Saldo posterior da conta 430: 34.278649
~~~~~~ Saldo posterior da conta 238: 1965.721351
```

Log de Execução

```
$$$$$$$Processando Transação T5 $$$$$$$$
~~~~~~ Saldo anterior da conta 454: 1000.000000
~~~~~~ Saldo anterior da conta 45: 1000.000000

=============>     445.44 transferido da conta:  454 para a conta: 45

~~~~~~ Saldo posterior da conta 454: 554.555022
~~~~~~ Saldo posterior da conta 45: 1445.444978


$$$$$$$Processando Transação T6 $$$$$$$$
~~~~~~ Saldo anterior da conta 186: 830.277195
~~~~~~ Saldo anterior da conta 314: 1000.000000

=============>     369.02 transferido da conta:  186 para a conta: 314

~~~~~~ Saldo posterior da conta 186: 461.255852
~~~~~~ Saldo posterior da conta 314: 1369.021343


$$$$$$$Processando Transação T7 $$$$$$$$
~~~~~~ Saldo anterior da conta 73: 1000.000000
~~~~~~ Saldo anterior da conta 23: 1000.000000

=============>     232.89 transferido da conta:  73 para a conta: 23

~~~~~~ Saldo posterior da conta 73: 767.110393
~~~~~~ Saldo posterior da conta 23: 1232.889607
```

Log de Execução

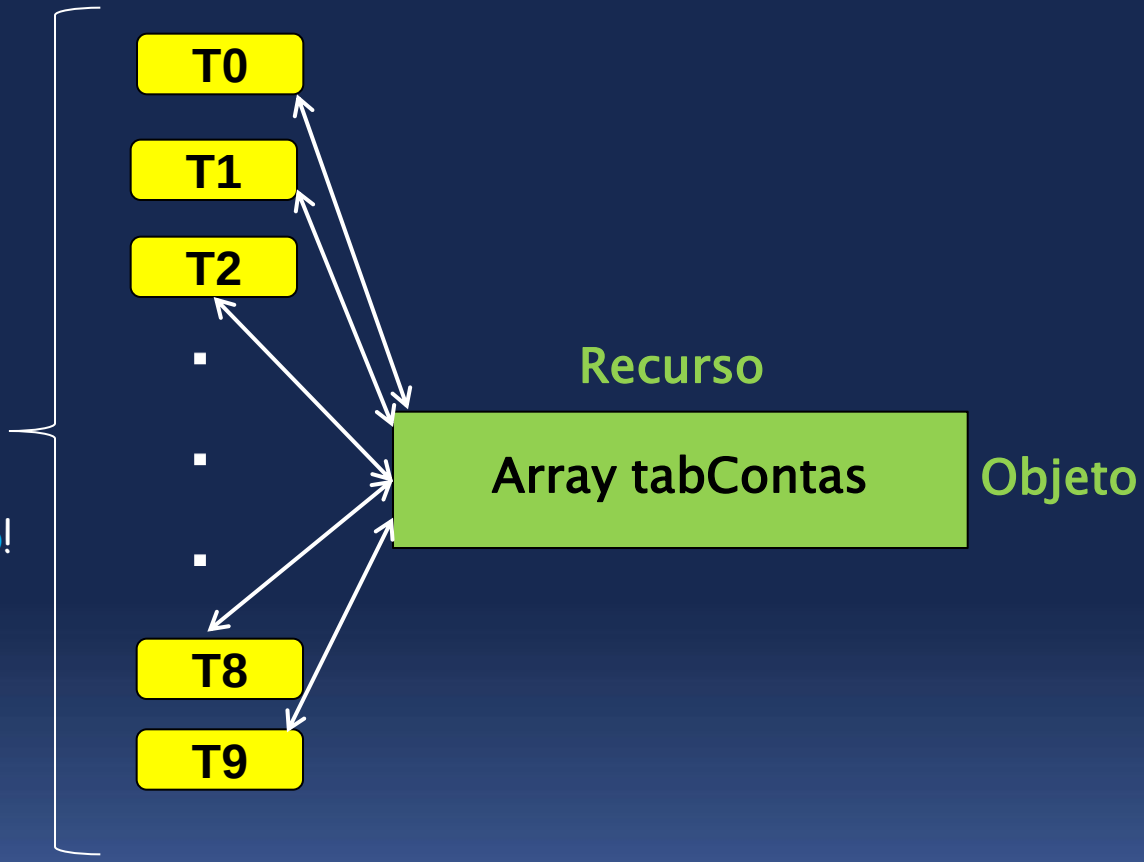
```
$$$$$$$Processando Transação T8 $$$$$$$$
~~~~~~ Saldo anterior da conta 485: 1000.000000
~~~~~~ Saldo anterior da conta 299: 1000.000000

=============>      14.79 transferido da conta:  485 para a conta: 299

~~~~~~ Saldo posterior da conta 485: 985.206632
~~~~~~ Saldo posterior da conta 299: 1014.793368


$$$$$$$Processando Transação T9 $$$$$$$$
~~~~~~ Saldo anterior da conta 360: 1000.000000
~~~~~~ Saldo anterior da conta 293: 1000.000000

=============>      56.00 transferido da conta:  360 para a conta: 293

~~~~~~ Saldo posterior da conta 360: 944.003708
~~~~~~ Saldo posterior da conta 293: 1055.996292
```

```
**** Saldo Total do Banco após as Transações: R$ 500000.00
```

Observações

- O processamento foi tipicamente **sequencial**, e a **log** de **execução** mostra que os valores finais da somatória de todos os saldos da contas concide com o valor inicial, atestando que o processamento foi executado **com sucesso**.

- As transações T0 até T9 acessaram o recurso tabContas de modo sequencial e o processamento foi **Correto**!



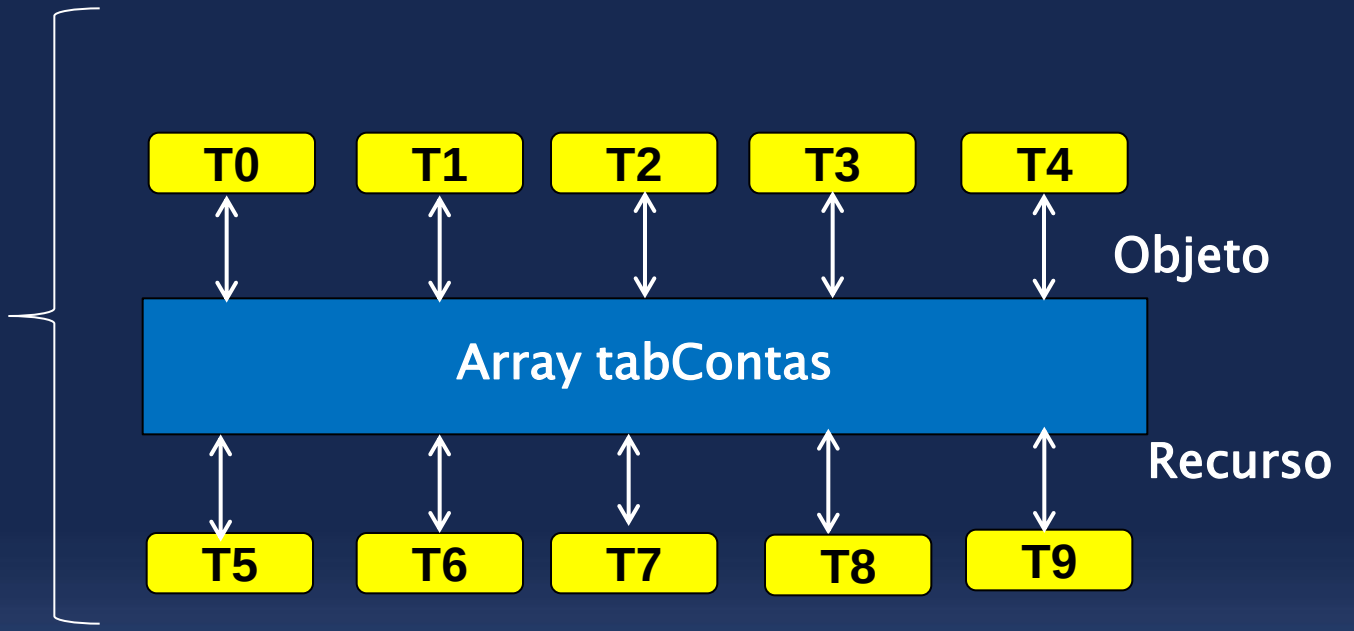
Que tal processarmos as transações de forma concorrente!

Numa situação real em um Banco é isso que ocorre !!!!

Convertendo para Threads

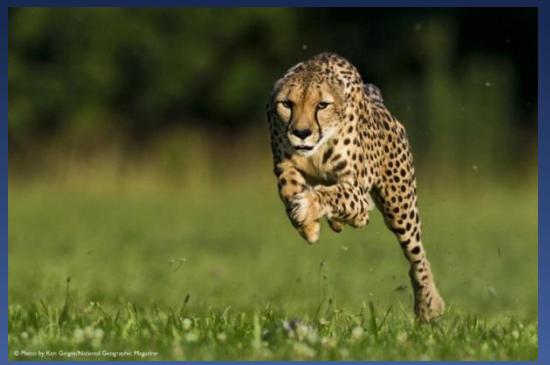
- As transações agora serão processadas em **paralelo**, todas compartilhando o objeto **tabContas** da Classe **Banco**!

- As transações **T0** até **T9** acessarão o recurso **tabContas** de modo **concorrente**!
- O recurso tabContas deverá ser compartilhado entre os threads!



Threads compartilhando recursos

- Uma das situações mais **comuns** em Programação Concorrente ocorre quando mais de um Thread de execução **compartilha** algum **recurso**;
- Em aplicações concorrentes, é comum que vários threads leiam ou escrevam no mesmo arquivo ou em uma conexão com o banco de dados;
- Esses recursos compartilhados podem provocar situações de **erro** ou **inconsistência** de dados;
- Recomenda-se fortemente que sejam implementados **mecanismos de controle** para evitar essas **anomalias**;



Seção Crítica

- Corresponde à algum **bloco** de **código** que acessa um recurso compartilhado e que **não** pode ser executado por mais de um thread, **ao mesmo tempo**.

- n processos concorrem para usar algum dado compartilhado
- Cada processo tem um trecho de código chamado seção crítica no qual os dados compartilhados estão sendo acessados.
- O problema: ter certeza que enquanto um processo está executando dentro da sua seção crítica, nenhum outro processo é permitido de executar na sua seção crítica.
- Estrutura de processo P_i

repeat

seção de entrada

seção crítica

seção de saída

seção restante

until *false*;

Seção Crítica

- Para auxiliar os programador a implementar **seções críticas**, linguagens concorrentes habitualmente fornecem **mecanismos de sincronização**;

- Quando um thread quer acessar uma **seção crítica**, ele usa um **mecanismo de sincronização** para saber se há algum outro thread rodando nessa **seção crítica**;

- Se **não** houver, o thread entra nessa **seção crítica** , caso contrário, o thread é **suspenso** pelo mecanismo de sincronização até que o thread que está processando na **seção crítica** termine;

- Quando mais de um thread estiver aguardando algum outro thread terminar a execução em uma **seção crítica**, a **JVM** escolhe um deles e o restante fica na fila aguardando sua vez.

Métodos de Sincronização em Java

- O método mais básico para se implementar sincronização em Java é o uso da keyword **synchronized** para controlar o acesso simultâneo à um método;
- Apenas um thread de execução irá acessar o método declarado com **synchronized**;
- Se algum outro thread tentar acessar qualquer método declarado com **synchronized** do mesmo objeto, ele será **suspenso** e ficará em estado **waiting**;
- Em outras palavras, cada método declarado com a palavra-chave **synchronized** corresponde à uma **seção crítica** e, assim, fica restrito à execução de um **único** thread.

Reescrevendo a aplicação

```
package br.uscs;

public class Banco {

    private double[] tabContas;
    private final static double maxTransfer = 1000.00;

    public Banco() {

        this.tabContas = new double[500];

        for (int i = 0; i < tabContas.length; i++) {
            tabContas[i] = 1000.0;
        }
    }
```

```
    public static double getMaxtransfer() {
        return maxTransfer;
    }

    public double[] getTabContas() {
        return tabContas;
    }

    public int size() {
        return tabContas.length;
    }

    public double getSaldoTotalBanco() {
        double saldoTotalBanco = 0;
        for (int i = 0; i < tabContas.length; i++ )
            saldoTotalBanco = saldoTotalBanco + tabContas[i];
        return saldoTotalBanco;
    }
}
```

```java
package br.uscs;

public class Transacao implements Runnable{
    private Banco b;

    public Transacao(Banco b) {
            this.b = b;
    }

    public void run() {

        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("########################" + Thread.currentThread());
        int contaOrigem = (int) (b.size() * Math.random());
        int contaDestino = (int) (b.size() * Math.random());

        double valorTransferencia = Banco.getMaxtransfer() * Math.random();
        this.transfer(contaOrigem, contaDestino, valorTransferencia);
    }
```

```
public synchronized void transfer(int contaOrigem ,int contaDestino,double valorTransferencia) {


    System.out.printf("%s ~~~~~~ Saldo anterior da conta %d: %f \n",Thread.currentThread(),
                      contaOrigem, b.getTabContas()[contaOrigem]);
    System.out.printf("%s~~~~~~ Saldo anterior da conta %d: %f \n",Thread.currentThread(),
                      contaDestino, b.getTabContas()[contaDestino]);

    if (b.getTabContas()[contaOrigem] < valorTransferencia )
        System.out.println("**** Saldo Insuficiente ****\n");
    else {
        b.getTabContas()[contaOrigem] = b.getTabContas()[contaOrigem] - valorTransferencia;
        b.getTabContas()[contaDestino] = b.getTabContas()[contaDestino] + valorTransferencia;

        System.out.printf("%s\n=============> %10.2f transferido da conta:  %d para a conta: %d\n\n",
                Thread.currentThread(),  valorTransferencia, contaOrigem, contaDestino);

        System.out.printf("%s~~~~~~ Saldo posterior da conta %d: %f \n", Thread.currentThread(),
                contaOrigem, b.getTabContas()[contaOrigem]);
        System.out.printf("%s~~~~~~ Saldo posterior da conta %d: %f \n\n\n", Thread.currentThread(),
                contaDestino, b.getTabContas()[contaDestino]);
    }
}

}
```