Lista 2

Na matemática, o número PI é uma proporção numérica definida pela relação entre o perímetro de uma circunferência e seu diâmetro;

É representado pela letra grega π. A letra grega π (lê-se: pi), foi adotada para o número a partir da palavra grega para perímetro, "περίμετρος", provavelmente por William Jones em 1706, e popularizada por Leonhard Euler alguns anos mais tarde.

Outros nomes para esta constante são constante circular ou número de Ludolph.

O π é um número irracional e o seu último cálculo (2013) chegou a 8.000.000.000.000.000 casas decimais.

Há diversas formas de se calcular o valor de π dentre estas, a do matemática John Wallis (1655).

$$\frac{2}{1}\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdot\frac{8}{7}\cdot\frac{8}{9}\cdot\cdots=\frac{\pi}{2}.$$

Implemente um programa em Java que faça o cálculo de Pi com o número de termos da aproximação definidas pelo usuário.

Gabarito

São duas classes, a que faz o cálculo propriamente dito e a que possui a main.

A que faz o cálculo propriamente dito deve: 1) Herdar a classe Thread; OU 2) Implementar a Runnable. Preferencialmente devemos sempre fazer de acordo com o segundo tipo, visando deixar o código mais eficiente.

A segunda é a classe que vai instanciar as threads e executá-las.

A primeira etapa é dividir o trabalho entre as threads. Observando a série, percebe-se que o termo geral é formado por duas frações. Cada termo será gerado dentro de um laço a partir de um índice. Considerando que iniciaremos o índice com 1, a primeira fração do termo geral é formado pela divisão do número **2*i** com o número **2*i-1**. A segunda fração será formada por **2*i** no numerador e dividido pelo número **2*i+1**.

Desta feita, a expressão geral é:

$$\frac{2*i}{(2*i-1)}\frac{2*i}{(2*i+1)} \quad (1)$$

A próxima etapa é separar as tarefas entre as threads. Em sala de aula fizemos isso dividindo o espaço do domínio em regiões consecutivas, ie, dividimos o espaço de domínio em, por exemplo, 4 regiões iguais e entregamos cada região para uma thread. A primeira thread recebia a primeira região, a segunda thread recebia a segunda e assim sucessivamente.

Neste exercício dividiremos de forma diferente. Usaremos o algoritmo de balanceamento de carga de **Round Robin**. Assim, se utilizarmos **N** threads, o primeiro termo fica com a primeira thread, o segundo com a segunda thread, e assim sucessivamente até o **N-1** termo ficará com a **N-1** thread. O termo de ordem **N** ficará com a primeira thread, novamente. Para isso, raciocinaremos que a quantidade de termos é um múltiplo da quantidade de threads que usaremos. Essa suposição é meramente para simplificar as nossas contas e, se não fosse múltiplo, ter-se-ia um fator adicional para alguma thread, o que não prejudicaria o processamento em si.

Para realizar esta distribuição, poderíamos usar a função módulo, conforme vimos em sala, porém aqui faremos diferente. Como já precisaríamos utilizar o **for**, nos faremos valer do campo de **incremento** desta estrutura e ao invés de fazermos de forma clássica (**i++**) faremos usando a quantidade de threads, ou seja, **i+=tamanho**, onde **tamanho** é a quantidade de threads que usaremos.

Assim a parte central da classe thread, que faz as contas, fica:

```
for(int i=this.passos; i<=this.n; i+=this.tamanho){
    double N = i;
    calculo *= 2.*N*2.*N/(2.*N-1)/(2.*N+1);
}
```

Onde:

Passos: indica o número da thread
N: indica a quantidade de termos que esta thread fará as contas
Tamanho: indica a quantidade de threads.

A próxima etapa corresponde em preparar a thread para fazer as contas. Usaremos um atributo estático para armazenar o valor final. Faremos isso apenas com o critério de simplificação uma vez que esse atributo, por ser estático, é visível por todas as instâncias[1]. Ainda, para assegurar que as threads iniciem e terminem juntas, vamos usar um sinalizador que cria uma barreira na execução do código até

[1] Caso o uso de atributos estáticos seja uma novidade, consulte: Java como Programar – Deitel & Deitel.

que todas as threads estejam naquele ponto de execução. Isto pode ser realizado através do método **await** da classe **CyclicBarrier**[2].

Assim, a parte central do código da thread fica:

```
1   this.barrier1.await();
2
3   double calculo = 1.;
4   for(int i=this.passos; i<=this.n; i+=this.tamanho){
5       double N = i;
6       calculo *= 2.*N*2.*N/(2.*N-1)/(2.*N+1);
7   }
8
9   this.pi *= calculo;
10  this.barrier2.await();
```

As linhas 1 e 10 possuem as barreiras para assegurar que que as threads comecem e terminem juntas.

As linhas de 4 a 6 possuem o cálculo dos termos, conforme explicando anteriormente.

A linha 9 armazena o cálculo realizado nesta thread no atributo estático.

A classe que instanciará as threads terá a interação com o usuário que informará a quantidade de termos e de threads que serão usadas. Além disso, é nessa classe que são instanciadas as barreiras que são usadas no código de cada thread.

Códigos finais:

1) Thread

```
public class pi implements Runnable{

    public int passos, n, tamanho;
    public static double pi=1;
    CyclicBarrier barrier1 = null;
    CyclicBarrier barrier2 = null;

    public pi(int passos, int n, int tamanho, CyclicBarrier barrier1,
CyclicBarrier barrier2){
        this.n = n;
        this.passos = passos;
        this.tamanho = tamanho;
        this.barrier1 = barrier1;
        this.barrier2 = barrier2;
    }
```

[2] Consulte Java como Programar – Deitel & Deitel, Cap 26, 9ª Edição.

```
    @Override
    public void run() {
        try {
            System.out.println("Iniciando a thread " +
Thread.currentThread().getName() );
            this.barrier1.await();

            double calculo = 1.;
            for(int i=this.passos; i<=this.n; i+=this.tamanho){
                double N = i;
                calculo *= 2.*N*2.*N/(2.*N-1)/(2.*N+1);
            }

            this.pi *= calculo;
            this.barrier2.await();

            System.out.println("Thread " +
Thread.currentThread().getName() + " finalizada");
            if(Thread.currentThread().getName().equals("Thread-0"))
System.out.println(2.*this.pi);

        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }

}
```

2) Programa principal

```
import java.util.ArrayList;
import java.util.Scanner;
import java.util.concurrent.CyclicBarrier;

public class PiComThreads {

public static void main(String[] args) {
        double pi=1;
        Scanner leitura = new Scanner(System.in);
        int nThreads, nFatores;
        System.out.print("Quantas threads:  ");
        nThreads = leitura.nextInt();
        System.out.print("Quantos fatores: ");
        nFatores = leitura.nextInt();
        Runnable barrier1Action = new Runnable() {
            @Override
            public void run() {
                System.out.println("Iniciar...");
            }
        };
        Runnable barrier2Action = new Runnable() {
            @Override
            public void run() {
                System.out.println("Finalizar...");
            }
        };

        CyclicBarrier barrier1 = new CyclicBarrier(nThreads,
barrier1Action);
        CyclicBarrier barrier2 = new CyclicBarrier(nThreads,
```

```
barrier2Action);

        ArrayList<pi> Pi = new ArrayList<>();

        for(int i=1; i<=nThreads; i++){
            pi barrierRunnable1 = new pi(i, nFatores/nThreads,
nThreads, barrier1, barrier2);
            new Thread(barrierRunnable1).start();
            Pi.add(barrierRunnable1);
        }

    }

}
```