

Java Avançado

Exercícios Propostos

Concorrência e Paralelismo com *java.util.concurrent*

www.softblue.com.br

# Exercício

Implemente uma corrida de sapos, onde cada sapo é representado por uma thread. Os sapos fazem basicamente duas ações: pulam e descansam, e repetem isto até o final do percurso. Para a disputa ficar mais emocionante, considere que o tamanho do pulo e o tempo de descanso são randômicos. No final, deve existir um ranking com a colocação de cada sapo. O número de sapos participantes, a distância e o intervalo da geração dos números randômicos fica a seu critério.

Na programação, utilize apenas as classes/interfaces dos pacotes java.util.concurrent e java.util.concurrent.locks. Não utilize a classe Thread ou o modificador synchronized.

# Exercício

A série de Gregory é uma fórmula matemática bastante simples que permite fazer o cálculo do valor de . Esta é a fórmula:

O cálculo de é iniciado através de uma somatória onde o valor de n varia de 0 até o infinito. Quanto maior o valor de n, mais preciso é o valor de (e mais “pesado” computacionalmente é o algoritmo). Ao final da somatória, basta multiplicar o valor por 4 e o valor de é encontrado.

A maneira mais simples de executar este cálculo é na forma sequencial. Mas ele também pode ser feito através de várias execuções simultâneas, onde o cálculo é dividido em tarefas e cada tarefa calcula uma parte do somatório. No final, os resultados das tarefas podem ser agrupados para obter o resultado final.

Crie uma aplicação que, usando a série de Gregory, faça o cálculo de usando múltiplas threads. Cada thread deve calcular o somatório para alguns valores de n e armazenar o resultado em um array compartilhado entre as threads. Assim que todas as threads finalizarem seus respectivos cálculos, a thread principal deve finalizar o cálculo e mostrar o valor calculado de na tela.

Na programação, utilize future tasks para fazer os cálculos parciais da série.

# Exercício

Implemente o clássico problema do produtor-consumidor. Este problema é composto por um produtor, um consumidor e um buffer compartilhado. O produtor e o consumidor executam de forma independente, onde o primeiro produz itens no buffer e o último consome itens do buffer. O problema é que o produtor não pode produzir se o buffer estiver cheio e o consumidor não pode consumir se o buffer estiver vazio.

Na programação, utilize apenas as classes/interfaces dos pacotes java.util.concurrent e java.util.concurrent.locks. Não utilize a classe Thread ou o modificador synchronized.

# Exercício

A sequência de Fibonacci é uma [sequência](http://pt.wikipedia.org/wiki/Sequ%C3%AAncia_(matem%C3%A1tica)) de [números naturais](http://pt.wikipedia.org/wiki/N%C3%BAmero_natural), na qual os primeiros dois termos são 0 e 1, e cada termo subsequente corresponde à [soma](http://pt.wikipedia.org/wiki/Soma) dos dois precedentes (fonte: <http://pt.wikipedia.org/wiki/N%C3%BAmero_de_Fibonacci>).

Portanto, a sequência é dada pelos seguintes números: **0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...**

Na matemática, a definição de um número da sequência de Fibonacci () é dada da seguinte forma:

* Se :
* Se :
* Se :

Estas três fórmulas mostram que o primeiro número da sequência () é o 0 e o segundo número da sequência () é o 1. A partir do terceiro número () os números da sequência são calculados somando os dois números anteriores da sequência.

A natureza recursiva do cálculo de um número da sequência de Fibonacci permite a utilização de um algoritmo recursivo para encontrar um determinado número da sequência. Veja um exemplo deste algoritmo escrito em Java:

public long fibonacci(int n) {

if (n <= 1) {

return n;

}

return fibonacci(n - 2) + fibonacci(n - 1);

}

Crie um programa que imprima que imprima os 30 primeiros números da sequência de Fibonacci, de forma que o cálculo de cada número deve ser feito utilizando programação paralela.

**Dica**: É possível dividir o cálculo de fibonacci(n - 2) e fibonacci(n - 1) em duas partes, as quais podem ser somadas depois para determinar o resultado final.