06

Criando a primeira Thread

Transcrição

Vamos ver uma outra aplicação, já que sabemos agora que a JVM usa Threads, que mostra um exemplo mais real. Nesse exemplo estamos criando uma aplicação Desktop com Swing para fazer um cálculo, uma simples adição. Mas antes, vamos importar as classes que utilizaremos, elas podem ser baixadas aqui (https://s3.amazonaws.com/caelum-online-public/threads1/calculador.zip).

Importamos duas classes: a primeira cria uma pequena tela com dois campos de textos (`JTextField`) e o botão (`JButton`) para executar uma multiplicação. Usamos aqui as classes da biblioteca Swing do Java para montar essa tela. Repare no código os `JTextField` , `JLabel` , o painel, e a janela.

```
public class TelaCalculador {

    public static void main(String[] args) {

        JFrame janela = new JFrame("Multiplicação Demorada");

        JTextField primeiro = new JTextField(10);
        JTextField segundo = new JTextField(10);
        JButton botao = new JButton(" = ");
        JLabel resultado = new JLabel("           ?          ");

        //quando clica no botão será executado a classe Multiplicador
        botao.addActionListener(new AcaoBotao(primeiro, segundo, resultado));

        JPanel painel = new JPanel();
        painel.add(primeiro);
        painel.add(new JLabel("x"));
        painel.add(segundo);
        painel.add(botao);
        painel.add(resultado);

        janela.add(painel);
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        janela.pack();
        janela.setVisible(true);
    }
}
```

Na segunda classe pegamos os valores dos dois campos de textos e fazemos a multiplicação:

```
public class AcaoBotao implements ActionListener {

    private JTextField primeiro;
    private JTextField segundo;
    private JLabel resultado;

    public AcaoBotao(JTextField primeiro, JTextField segundo, JLabel resultado) {
```

```
            this.primeiro = primeiro;
            this.segundo = segundo;
            this.resultado = resultado;
        }

        @Override
        public void actionPerformed(ActionEvent e) {

            long valor1 = Long.parseLong(primeiro.getText());
            long valor2 = Long.parseLong(segundo.getText());
            BigInteger calculo = new BigInteger("0");

            for (int i = 0; i < valor1; i++) {
                for (int j = 0; j < valor2; j++) {
                    calculo = calculo.add(new BigInteger("1"));
                }
            }

            resultado.setText(calculo.toString());
        }
    }
```

Só que essa multiplicação foi implementada de maneira muito muito ineficiente e por isso demora. Na nossa aplicação, ao calcular com valores menores não há problema! A multiplicação funciona, mas assim que colocarmos valores um pouco maiores, a nossa tela trava. Travar significa que o usuário não pode mais mexer nos campos de textos, a tela fica congelada. Ou seja, enquanto estamos trabalhando no cálculo ninguém pode mexer na tela. Isso é um comportamento ruim pensando no nosso usuário final, pois queremos que as coisas funcionem em paralelo. E aí entram os nossos **Threads**. O nosso objetivo então é criar um Thread próprio e executar a multiplicação em paralelo. Vamos lá?

A classe Thread

Criar um `Thread` é fácil, basta instanciar um objeto da classe `Thread` :

```
Thread threadMultiplicador = new Thread();
```

Mas de alguma forma é preciso dizer o que o `Thread` deveria fazer, no nosso caso é a multiplicação, certo? Nós precisamos passar a multiplicação para o `Thread` ! Olhando no construtor do `Thread` , podemos ver que a classe recebe algo que se chama o `Runnable` . Ele recebe algo que é rodável!

A interface Runnable

Essa `Runnable` é uma interface que possui apenas um método `run` . Nesse método vamos definir o que queremos executar nesse `Thread` que é justamente o cálculo de multiplicação. Então mãos à obra, vamos criar uma nova classe que implementa essa interface:

```
package br.com.alura.threads;

public class TarefaMultiplicacao implements Runnable {

    @Override
    public void run() {
```

```
            //esse método o nosso thread executará
        }

    }
```

O nosso `Thread` vai executar o método `run`, então falta implementar aquele cálculo demorado dentro desse método:

Vamos fazer com que a classe `TarefaMultiplicacao` receba os dois campos de texto e o botão, assim ela poderá fazer o cálculo. Então vamos recortar todo o código do método `actionPerformed`, da classe `AcaoBotao`, e colocar dentro do método `run`:

```
public class TarefaMultiplicacao implements Runnable {

    private JTextField primeiro;
    private JTextField segundo;
    private JLabel resultado;

    public TarefaMultiplicacao(JTextField primeiro, JTextField segundo,
JLabel resultado) {
        this.primeiro = primeiro;
        this.segundo = segundo;
        this.resultado = resultado;
    }

    @Override
    public void run() {

        long valor1 = Long.parseLong(primeiro.getText());
        long valor2 = Long.parseLong(segundo.getText());
        BigInteger calculo = new BigInteger("0");

        for (int i = 0; i < valor1; i++) {
            for (int j = 0; j < valor2; j++) {
                calculo = calculo.add(new BigInteger("1"));
            }
        }

        resultado.setText(calculo.toString());
    }
}
```

Perfeito, agora só falta passar um objeto desta classe para o nosso `Thread`, na classe `AcaoBotao`:

```
public void actionPerformed(ActionEvent e) {

    TarefaMultiplicacao tarefa = new TarefaMultiplicacao(primeiro, segundo, resultado);
    Thread threadMultiplicador = new Thread(tarefa);
}
```

Por fim, para o `Thread` realmente começar a trabalhar em paralelo é preciso inicializá-lo explicitamente. Para tal existe o método `start()`:

```
public void actionPerformed(ActionEvent e) {

    TarefaMultiplicacao tarefa = new TarefaMultiplicacao(primeiro, segundo, resultado);
    Thread threadMultiplicador = new Thread(tarefa);

    //thread começa a trabalhar
    threadMultiplicador.start();
}
```

Ótimo, agora já podemos testar o nosso código! Ao rodar a aplicação e fazer um cálculo demorado podemos ver que aplicação não trava mais. Os campos de texto continuam acessíveis, mesmo se o cálculo não terminou ainda.