

o que você quer aprender:

ENTRAR

MATRICULE-SE

TODOS OS
CURSOS

NOSSAS
FORMAÇÕES

PARA
EMPRESAS

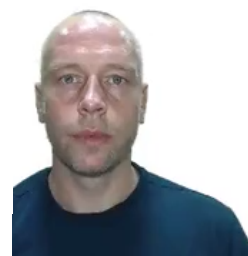
DEV
EM <T>

Alura > Cursos de Programação > Cursos de Java > Conteúdos de Java >
Primeiras aulas do curso Threads em Java 1: programação paralela

Threads em Java 1: programação paralela

Introdução à Threads - Introdução às Threads

0:00 / 3:00



Oi alunos, bem-vindo à nosso curso sobre Threads! Quem já ouviu falar sobre Threads, sabe que eles tem alguma coisa a ver com a execução paralela. Isso é muito importante hoje em dia. Por quê? Na verdade, usando o nosso computador, a gente passa o tempo todo fazendo várias coisas ao mesmo tempo.

impede que eu navegue por algum site, enquanto baixo o arquivo. Ele consegue executar diferentes tarefas simultaneamente.

Por que existem Threads?

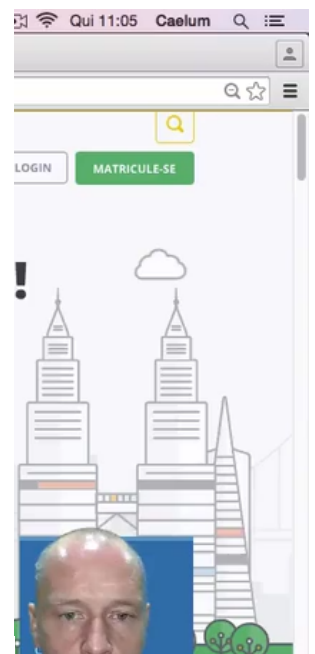
Agora, o nosso foco aqui é a plataforma Java. E quando você usa Java, você estará usando automaticamente Threads, porque o Java já nasceu com essa ideia de executar tarefas em paralelo. Por exemplo, neste momento com certeza, enquanto você está assistindo esse vídeo, tem outros alunos acessando outros cursos, fazendo exercícios, pesquisando ou trabalhando no fórum. Ou seja, o servidor precisa executar várias tarefas ou várias requisições ao mesmo tempo.

Se o servidor da Alura não pudesse trabalhar em paralelo, nossa plataforma seria muito lenta. O mesmo raciocínio pode ser aplicado para o banco de dados que a Alura usa. Com certeza, se o banco de dados não suportasse Threads, tudo seria muito mais lento. Então executar tarefas em paralelo é relacionado com o desempenho.

Nesse curso vamos ver os Threads de bastante baixo nível. Talvez, no seu dia a dia, você não precisaria de todo esse conhecimento detalhado, no entanto isso é muito útil para saber como as bibliotecas funcionam por debaixo dos panos, e até para entender as configurações delas. Um conhecimento básico faz parte do desenvolvedor.

Dica: existe um curso na Alura sobre as [Collections](#)!

Introdução à Threads - Visualizando a primeira Thread



Visualizando o primeiro Thread

E vou provar para vocês a existência de threads através de um pequeno programa. Vamos criar um novo projeto Java dentro do Eclipse, chamado **threads**, e criar uma nova classe `Principal` (no pacote `br.com.alura.threads`) e dentro dela um método `main`. Nós sabemos que qualquer programa Java sempre começa com o método `main`. Quando rodamos, automaticamente a máquina virtual irá criar um novo thread ou uma nova linha de execução. E essa linha de execução pode ser executada em paralelo!

O código é bem simples:

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        System.out.println("Thread MAIN");  
    }  
}
```

No método `main`, definimos apenas o que queremos executar, como a máquina virtual realmente faz esse trabalho, não sabemos. Como esse programinha acaba muito rápido, vamos pedir para a máquina virtual mandar aquele thread dormir! Como? Muito fácil, através do método estático `Thread.sleep()`:

```
public class Principal {  
  
    public static void main(String[] args) throws InterruptedException  
  
        System.out.println("Thread MAIN");  
  
        Thread.sleep(50000);  
    }  
}
```

ferramenta jconsole na linha de comando.

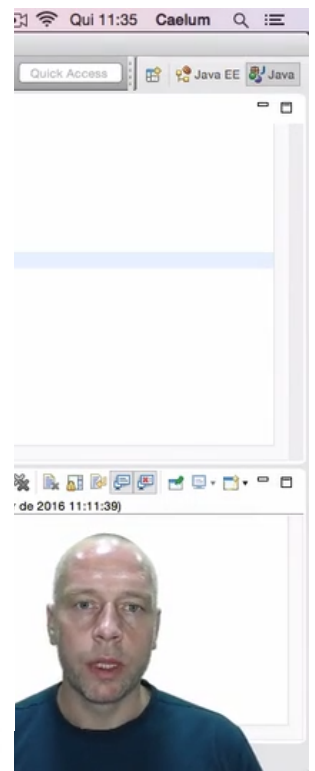
Se você não tem instalada essa ferramenta não se preocupe, ela não é necessária para continuação do treinamento.

Através dela podemos nos conectar como nossa aplicação. A ferramenta jconsole mostra informações sobre a máquina virtual que está rodando o nosso programa.

Reparem que existe uma aba com o nome *Threads* e nessa aba podemos ver que já existem vários threads, inclusive um que se chama **main**. Esse é aquele que rodou o nosso método `main`! Então a máquina virtual já cria vários Threads por padrão (por exemplo para atender conexões remotas ou fazer a coleta de lixo). Isso já está embutido na JVM e foi grande diferencial quando Java foi lançado.

Introdução à Threads - Criando a primeira Thread

0:01 / 16:07



Criando o primeiro Thread

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](#).

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, resul-  
  
        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:

```
private JTextField primeiro;  
private JTextField segundo;  
private JLabel resultado;  
  
public AcaoBotao(JTextField primeiro, JTextField segundo, JLabel r  
    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?

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:

```
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, segun
```


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);  
    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.

Sobre o curso Threads em Java 1: programação paralela

O [curso Threads em Java 1: programação paralela](#) possui **143 minutos de vídeos**, em um total de **63 atividades**. Gostou? Conheça nossos outros [cursos de Java](#) em [Programação](#), ou leia nossos [artigos de Programação](#).

Matricule-se e comece a estudar com a gente hoje! Conheça outros tópicos abordados durante o curso:

- Introdução à Threads
- Ordem de execução
- Sincronizando a execução
- Coleções Thread Safe
- Espere e notifique
- Revisitando a lista
- Entendendo Deadlock