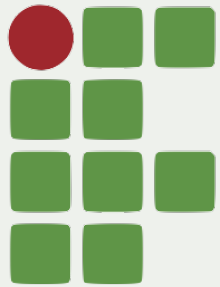
APIs Java: Threads
Programação Concorrente

POO29004 – Engenharia de Telecomunicações

Prof. Emerson Ribeiro de Mello

http://docente.ifsc.edu.br/mello/poo

12 de junho de 2020



INSTITUTO FEDERAL
Santa Catarina
Câmpus São José

Aplicação com uma única Thread (linha de execução)

```
1  public class Fluxo1{
2     public void disparar(){
3        for(int i=0; i<10;i++){
4           System.err.println("Fluxo 1");
5        }
6     }
7  }
8  public class Fluxo2{
9     public void disparar(){
10       for(int i=0; i<10;i++){
11          System.err.println("Fluxo 2");
12       }
13    }
14 }
15 public class Principal{
16    public static void main(String[] args){
17       Fluxo1 f1 = new Fluxo1();
18       Fluxo2 f2 = new Fluxo2();
19       f1.disparar();
20       f2.disparar();
21       System.err.println("Fim do programa");}
22 }
```

Programação concorrente – sistemas multitarefas

- Sistemas operacionais modernos são caracterizados como **multitarefas**, pois executam **diversos processos simultaneamente**
- Cada processo possui suas próprias variáveis e a troca de informações (comunicação) entre processos é feita por meio de arquivos em disco ou *sockets*

Permite que um processo realize diversas tarefas de forma concorrente

- Uma *thread* fica responsável por interagir com o usuário (ler de teclas)
- Outra *thread* fica responsável por escrever em um *sockets* de rede

- Por estarem dentro de um mesmo processo, compartilhando variáveis, a comunicação entre *threads* tende a ser mais eficiente e mais fácil de ser feita do que a comunicação entre dois processos distintos

Desenvolvendo aplicação multithread em Java

- Para desenvolver uma aplicação *multithread* é necessário
  1. Escrever o código que será executado por cada *Thread*
  2. Escrever o código que irá disparar uma ou mais *Thread*

Desenvolvendo aplicação multithread em Java

- Para desenvolver uma aplicação *multithread* é necessário
  1. Escrever o código que será executado por cada *Thread*
  2. Escrever o código que irá disparar uma ou mais *Thread*

Em Java é possível criar uma Thread de duas formas

1. Criar uma classe que estenda a classe **Thread**
   - Deve-se sobrescrever o método `public void run()`
2. Criar uma classe que implemente a interface **Runnable**
   - Deve-se implementar o método `public void run()`
   - Opção interessante já que Java não possui o conceito de herança múltipla

Threads em Java

- Herança

```
23 public class Fluxo1 extends Thread{
24    public void run(){
25       for(int i=0; i<10;i++){
26          System.err.println("Fazendo uso de herança");
27       }
28    }
29 }
```

- Interface

```
30 public class Fluxo2 implements Runnable{
31    public void run(){
32       for(int i=0; i<10;i++){
33          System.err.println("Fazendo uso de interface");
34       }
35    }
36 }
```


Exemplo de uso com herança e interface

```
71 public static void main(String[] args){
72   Thread comHeranca = new Fluxo1();
73   Thread comInterface = new Thread(new Fluxo2());
74
75   //executando as threads
76   comHeranca.start();
77   comInterface.start();
78
79   System.err.println("Fim do programa");
80 }
```


Exemplo de uso com herança e interface

```
71 public static void main(String[] args){
72   Thread comHeranca = new Fluxo1();
73   Thread comInterface = new Thread(new Fluxo2());
74
75   //executando as threads
76   comHeranca.start();
77   comInterface.start();
78
79   System.err.println("Fim do programa");
80 }
```

O que será impresso na tela?

1. 10 linhas com herança + 10 linhas com interface + Fim do programa
2. Fim do programa + 10 linhas com herança + 10 linhas com interface
3. Não tenho como prever

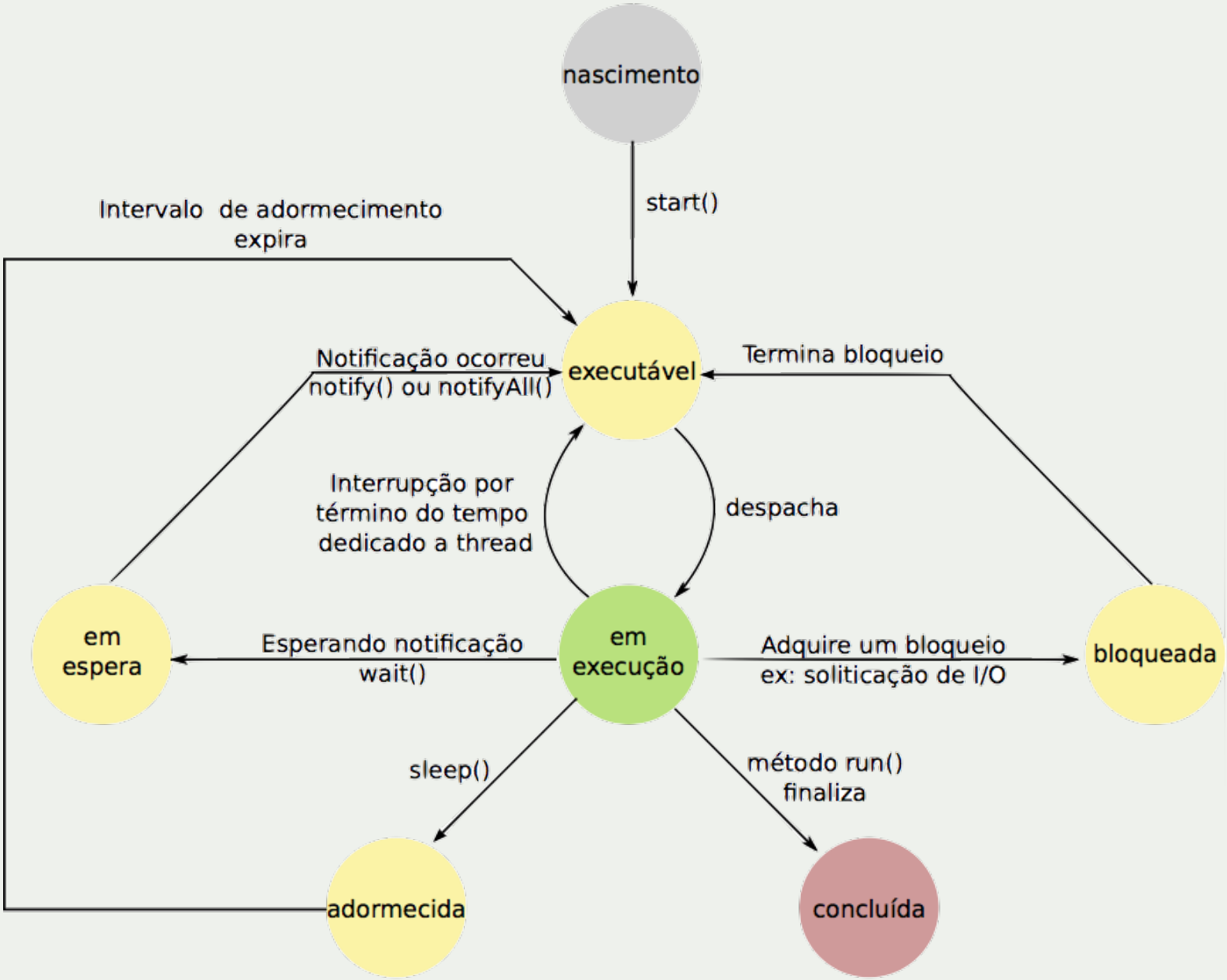
Exemplo de uso com herança e interface

```
71 public static void main(String[] args){
72   Thread comHeranca = new Fluxo1();
73   Thread comInterface = new Thread(new Fluxo2());
74
75   //executando as threads
76   comHeranca.start();
77   comInterface.start();
78
79   System.err.println("Fim do programa");
80 }
```

O que será impresso na tela?

1. 10 linhas com herança + 10 linhas com interface + Fim do programa
2. Fim do programa + 10 linhas com herança + 10 linhas com interface
3. Não tenho como prever

Ciclo de vida de uma thread



Principais métodos para trabalhar com threads

start	Ocorre a invocação do método **run** da Thread. ■ Após disparar a *thread*, o fluxo de execução retorna para o seu chamador imediatamente
run	Onde é colocada a lógica do fluxo ■ Ao finalizar este método, a *thread* morre
sleep	Faz com que a *thread* durma por alguns milisegundos ■ **Importante**: Enquanto uma *thread* dorme, ela não disputa o processador ■ Exemplo de uso dentro do método **run**: `Thread.sleep( 1000 );`
join	Espera que a *thread* que fora invocada morra antes de retornar para a *thread* que a invocou

Fazendo a thread dormir por 1000 milisegundos

```
81 public class Fluxo3 extends Thread{
82
83    public Fluxo3(String nome){
84       super(nome);
85    }
86
87    public void run(){
88       try{
89
90          System.err.println( this.getName() + " vair dormir...");
91          Thread.sleep(1000);
92
93       }catch(InterruptedException e){
94          System.err.println(e.toString());
95       }
96       System.err.println(this.getName() + " acordou...");
97    }
98 }
```


Exemplo de uso do método join

```
99  public static void main(String[] args){
100   Thread f3 = new Fluxo3();
101
102   //disparando a thread
103   f3.start();
104   System.err.println("Depois do start e antes do join");
105   try{
106
107     f3.join();
108     // a linha abaixo e' executada somente depois
109     // de finalizar o metodo run do objeo f3
110     System.err.println("Depois do join");
111
112   }catch(InterruptedException ex) {
113     System.err.println(ex.toString());
114   }
115
116   System.err.println("Fim do programa");
117 }
```

Memória compartilhada entre *threads*

- Somente uma thread por vez pode acessar uma memória compartilhada (i.e. uma variável)

- Java implementa o conceito de `monitor` para impor o acesso mutuamente exclusivo aos métodos
  - Tais métodos devem apresentar a palavra **synchronized**
- Quando um método sincronizado é executado o monitor é consultado
  - Se não existir outro método sincronizado em execução, então continua; Senão, aguarde pela notificação
- Métodos para trabalhar com sincronismo:
  - **wait**, **notify** e **notifyAll**;

Exemplo: Barbearia com várias *threads*

- Baixe o código disponível em https://github.com/poo29004/barbearia-threads

Exercícios

Exercício 1: Olá mundo reverso

- Escreva um programa chamado **OlaMundoReverso** o qual deverá disparar uma *thread*, chamada Ola-01. A *thread* Ola-01 deve criar a *thread* Ola-02. A *thread* Ola-02 deve criar a *thread* Ola-03 e assim sucessivamente até criar a *thread* Ola-10.
- Cada *thread* deverá imprimir “Olá mundo, sou a *thread* Ola-XX”, contudo essa mensagem de saudação deve ser impressa na ordem inversa da criação da *thread*. Ou seja, a *thread* Ola-10 deve ser a primeira a imprimir a mensagem e a *thread* Ola-01 deverá ser a última a imprimir.

Exercício 2: Ray tracing

- *Ray tracing* é um algoritmo usado para renderização de imagens tridimensionais e baseia-se na simulação do trajeto que os raios de luz percorreriam no mundo real. Aqui tem um tutorial sobre.
- O código disponível aqui faz uso da biblioteca `raytracer.jar` para gerar uma sequência de imagens

```
118 javac -cp raytracer.jar:.  MakeMovieSequential.java
119
120 java -cp raytracer.jar:.  MakeMovieSequential 30
121
122 convert -delay 6 frame_00*.png filme.mpg
```

Seria possível gerar mais rapidamente a sequência de imagens? Faça as modificações necessárias no código fornecido

Como adicionar uma biblioteca (arquivo .jar) no projeto com gradle

- Criar um diretório `lib` na raiz do seu projeto
- Baixar o arquivo `raytracer.jar` e salvar dentro do diretório lib
- Adicione a biblioteca como dependência no arquivo `build.gradle`

```
123 dependencies {
124     testCompile group: 'junit', name: 'junit', version: '4.12'
125
126     // importando arquivo JAR que está dentro do diretório lib
127     compile files('lib/raytracer.jar')
128 }
```

Ajuste no .gitignore

Para manter o raytracer.jar no projeto, adicione a seguinte linha no final do arquivo `.gitignore`:

```
!lib/raytracer.jar
```


Leitura obrigatória

- HOSRTMANN, C. S., CORNELL, G. **Core Java** - 8 ª Edição, 2010. Capítulo 11.
- https://docs.oracle.com/javase/tutorial/essential/concurrency