Programação III

Aula 7 – Multithreading

Introdução

O que é uma Multithreading?

Permite a execução de múltiplas tarefas simultaneamente

Cada thread representa um fluxo independente de execução.

Benefícios

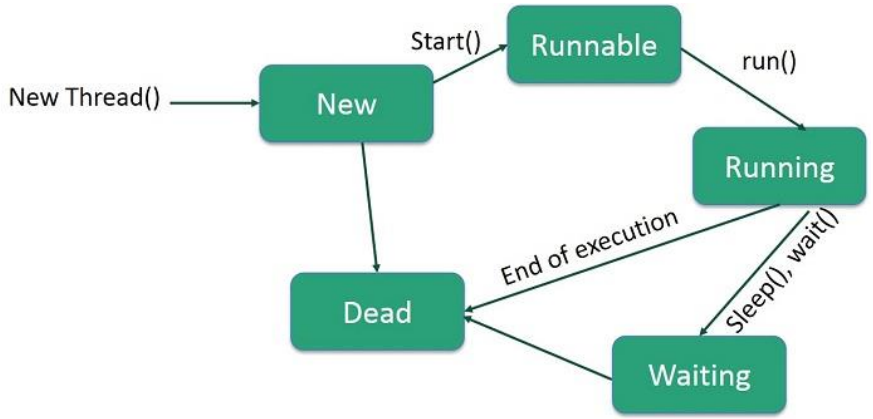
Melhor aproveitamento da CPU

Redução do tempo ocioso

Melhor responsividade em aplicações interativas

Modelo de Thread no Java

- **Como Java implementa threads?**
  - Java Runtime gerencia threads.
  - Threads são usadas internamente pelo Java (ex: Garbage Collector).
  - Eliminam o uso de "event loops" e polling excessivo.



- **Estados de uma Thread**
  - **NEW**: Criada, mas não iniciada.
  - **RUNNABLE**: Pronta para execução.
  - **BLOCKED**: Aguardando um recurso.
  - **WAITING**: Aguardando notificação.
  - **TIMED_WAITING**: Aguardando por um tempo determinado.
  - **TERMINATED**: Finalizada.

Ver MultithreadingExample.java

Criando Threads em Java

1 - Implementando Runnable

```
class MyThread implements Runnable {
     public void run() {
     System.out.println("Thread executando");
     }
}

class MyThread2 implements Runnable {
     public void run() {
     System.out.println("Thread 2 executando");
     }
}
```

```
class Demo {
     public static void main(String args[]) {
       Thread t = new Thread(new MyThread());
       Thread t2 = new Thread(new MyThread2());
       t.start();
       t2.start();
     }
}
```

Vantagens: permite herança de outra classe.

Criando Threads em Java

2 – Estendendo a classe Thread

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread executando");
    }
}
class Demo {
    public static void main(String args[]) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

Vantagens: mais intuitivo, mas restringe herança.

Controle de Threads

- **Principais Métodos da Classe Thread**
  - `getName()`, `setName()`
  - `getPriority()`, `setPriority()`
  - `sleep(ms)`, `yield()`
  - `isAlive()`
  - `join()` (espera uma thread terminar)

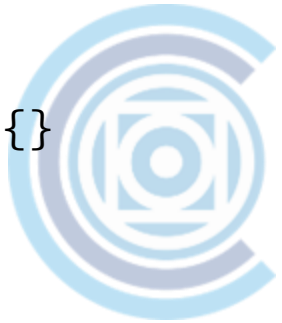
```
class Demo {
public static void main(String args[]) {

    Thread t1 = new Thread(() -> {
        try { Thread.sleep(5000); } catch
        (InterruptedException e) {}
        System.out.println("Thread finalizada");
    });

    try {
      t1.start();
      t1.join(); // Aguarda t1 terminar antes de
    continuar
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }

    }
}
```


Ver ThreadManagement.java | IoTExample.java | YieldDemo.java | DownloadMonitor.java | Banco.java

Sincronização de Threads

- **Problema de Acesso Concorrente**
  - Múltiplas threads acessando um recurso compartilhado podem gerar inconsistências.
  - Exemplo de *race condition* em um contador compartilhado.
- Uso de `synchronized`

```
class Shared {
    synchronized void show(String msg) {
      System.out.print("[" + msg);
      try { Thread.sleep(1000); } catch (InterruptedException e) {}
      System.out.println("]");
    }
}
```



Ver SemSincronizacao.java | ComSincronizacao.java | BancoDeDados.java | BancoTest.java

Comunicação entre Threads

- **Problema do Polling**
  - Threads podem consumir CPU desnecessariamente esperando eventos.
- **Métodos `wait()`, `notify()`, `notifyAll()`**
  - `wait()`: Thread dorme até `notify()` ser chamado.
  - `notify()`: Acorda uma única thread.
  - `notifyAll()`: Acorda todas as threads.

```
synchronized void get() {
  while (!disponivel)
  try {
    wait();
    System.out.println("Consumindo");
    disponivel = false;
    notify();
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```


Ver ProdutorConsumidor.java

Deadlocks

- Ocorre quando duas threads esperam indefinidamente uma pela outra.
- Exemplo:

```
class Recurso {
  synchronized void metodo(Recurso r) {
    r.metodo(this);
  }
}
```

Ver DeadlockExemplo.java e EvitaDeadlock.java