Programação Orientada a Objetos

Marcos Lapa dos Santos
marcoslapa@gmail.com

Threads

POO

Threads

- O conceito de **thread** está intimamente ligado ao conceito de **processo**.
- Um processo é basicamente um programa em execução constituído do código executável, dos dados referentes ao código, da pilha de execução, do valor do contador de programa (registrador PC), do valor do apontador de pilha (registrador SP) e dos valores dos demais registradores do hardware.

Estácio | FIB

POO

Threads

- Vários processos podem compartilhar o mesmo processador
- Os processos se revezam (o sistema operacional é responsável por esse revezamento) no uso do processador
- Para permitir esse revezamento é necessário salvar o contexto do processo que vai ser retirado do processador, para que futuramente o mesmo possa continuar sua execução (**voltando ao ponto exato de sua parada!**).

Estácio | FIB

POO

Threads

- Para entendermos o que são Threads é necessário que entendamos também o conceito de **Multitarefa**:
  - **Multitarefa é:** A capacidade de ter mais de um programa funcionando de forma “simultânea” em uma mesma máquina (a multitarefa **real** só ocorre em computadores com mais de um processador).
- Já os programas MultiThreads, ampliam a idéia da multitarefa

Estácio | FIB

POO

Threads

- Uma Thread, ou processo leve, é considerada a unidade básica de utilização da CPU
- No caso de programas MultiThreads, muitas linhas de execução *(threads)* podem ser criadas em um mesmo processo (elas pertencem ao processo).
- Essas threads, executam de forma “paralela” e, ao contrário dos múltiplos processos (multitarefa), elas compartilham as mesmas variáveis.

Estácio | FIB

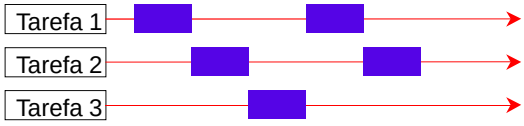
Multitarefa vs Multithread

- Vantagens das Threads
  - Threads são mais leves do que processos (elas estão contidas nos processos);
  - A comunicação entre processos é mais lenta e restritiva;
- Problema
  - A programação com Threads pode se tornar mais complexa e arriscada pois existe um compartilhamento de dados.

Execução de Threads

Várias threads em várias CPUs

Várias threads compartilhando uma única CPU



Exemplos (Multithread/Multitarefa)

- Um navegador deve tratar com vários hosts;
- Abrir uma janela de correio eletrônico ou ver outra página enquanto faz downloads;
- Editor de Texto: Enquanto se envia para a impressora um texto, pode-se continuar digitando o texto;
- UM servidor de bate papo (chat) que aceita vários clientes

A Classe java.lang.Thread

- Serve para construir uma nova thread em qualquer ponto do seu código
  - Instâncias desta classe, ou de seus descendentes, poderão ter seus códigos processados em paralelo
- Métodos principais:
  - **void run()** – método que deve ser sobrecarregado para incluir o código a ser processado em paralelo; Não deve ser chamado diretamente, pois não cria uma nova linha de execução no processo!
  - **void start()** – prepara uma linha de execução para ser iniciada invocando o método run() posteriormente.
  - **static void sleep(long ms)** – coloca a thread corrente em estado de suspensão por tantos milisegundos (parâmetro ms).

Prática 1

- Fazer um programa em Java que imprima na tela as strings: “Olá!” e “Threads!” 500 vezes cada uma, **alternadamente**.

- Usar agora o conceito de Threads:
  - Criar uma classe **MsgThread**, herdando da classe **Thread**.
  - Criar um **construtor** que receba um parâmetro **String** (mensagem) e guarde este valor em um atributo interno da classe chamado **msg**.
  - Implementar o método **run** imprimindo o valor do atributo interno **msg** 500 vezes.
  - A aplicação principal deverá instanciar dois objetos do tipo **MsgThread** passando uma mensagem diferente para cada thread.
  - Disparar as **threads** através do método **start()** dos objetos da classe **MsgThread** na aplicação principal.

- Observe o comportamento do seu programa agora!

Implementação de Threads

Implementação de Threads

- Podemos criar Threads em java de duas formas:
  - através da herança da classe **Thread** (Vimos na prática anterior)
  - ou através da implementação da interface **Runnable**
- Nos dois casos a funcionalidade (programação) das threads é feita na implementação do método **run**.
- A vantagem do uso da Interface Runnable está em permitir que sua classe herde de outra e se comporte como uma thread ao mesmo tempo.

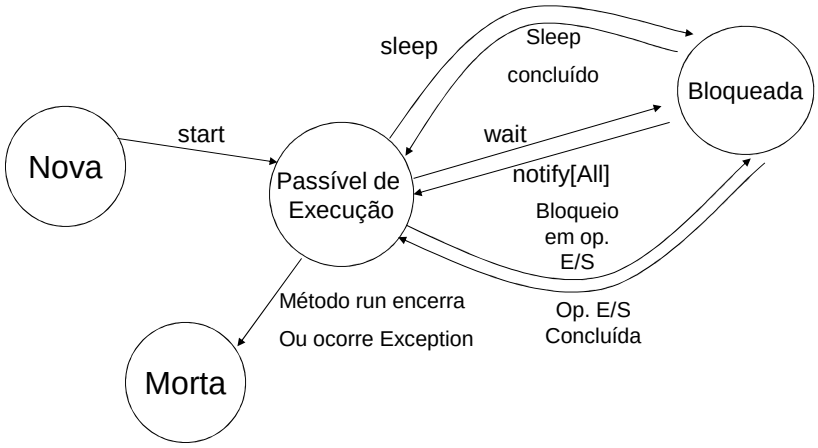
Implementando com a Interface Runnable

- A classe que implementa a thread no estilo Runnable deve declarar **um objeto** do tipo Thread e para instanciá-lo deve chamar o construtor da classe Thread passando como parâmetro a instância da própria classe que implementa Runnable e o nome da thread (opcional).
- Ex:

```
class ContaPoupanca extends Conta implements Runnable {
  private Thread thread1;
  ...
  public void iniciar() {
    if (thread1 == null){
      thread1 = new Thread(this, "NomeThread");
      thread1.start();
  }
}
```

Propriedades das Threads

- Estados possíveis para uma thread:



Estados possíveis das Threads

- Threads **Novas** – Thread que foram instanciadas e ainda não executadas - **método start().**

- Threads **Passíveis de Execução** – Assim que o método start() de uma thread é chamado, ela entrará neste estado, o qual **não significa que ela já esteja em execução**. A plataforma Java **não diferencia** os estados **Passível de Execução** e **Executando**.

Estados possíveis das Threads

- Threads **Bloqueadas** – ocorrem quando:
  - O método sleep foi chamado (Condição de saída: o tempo expirará em ***n*** milisegundos)
  - Bloqueio de I/O dentro da linha de execução (Condição de saída: operação de I/O terminar)
  - A thread chamou o método wait (Condição de saída: outra thread deverá chamar o notify ou notifyAll)
  - A thread tenta bloquear um objeto que já está bloqueado por outra (Condição de saída: a outra thread cede o bloqueio)

Estados possíveis das Threads

- Threads **Mortas** – ocorrem quando:
  - Morte natural (encerramento do método **run**)
  - Morte abrupta (ocorrência de alguma **exceção não tratada** no código)

POO

Prioridade

- O escalonamento é fundamental quando é possível a execução paralela de threads
- Certamente existirão mais threads a serem executadas que processadores
- Assim a execução paralela de threads é simulada através de mecanismos do escalonamento dessas threads

Estácio | FIB

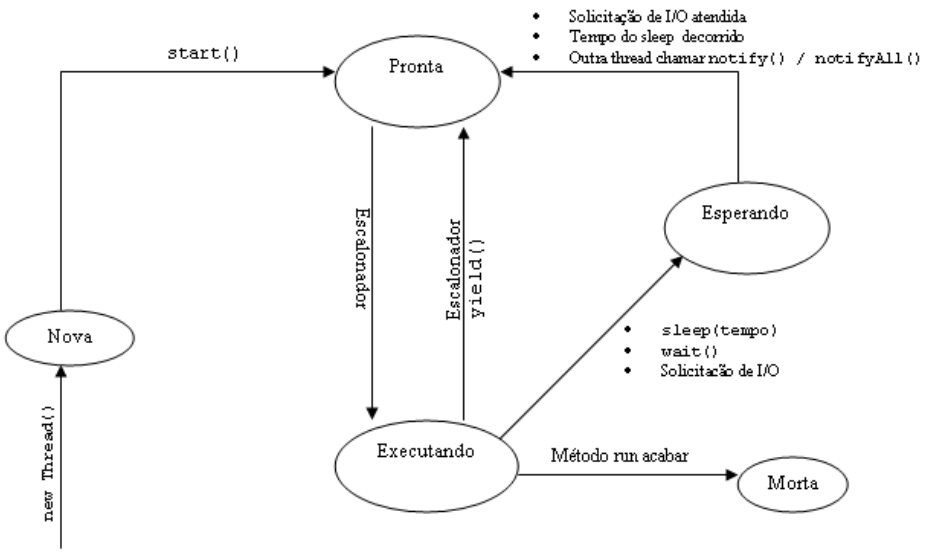
POO

Prioridade

- Toda thread possui uma prioridade, a qual pode ser um valor inteiro no intervalo [MIN_PRIORITY ... MAX_PRIORITY], (estas constantes estão definidas na classe Thread)
- Quanto maior o valor do inteiro maior a prioridade da thread.
- Cada thread Nova recebe a mesma prioridade da thread que a criou
- A prioridade de uma thread pode ser alterada através do método setPriority(int priority).

Estácio | FIB

POO



Estácio | FIB

POO

Prática 2

- Retorne ao código da Prática 1 e modifique o código para que suas classes que herdam de Thread agora implementem a interface Runnable.
- Instancie três threads com mensagens distintas agora usando o conceito de Runnable.
- Execute o código e analise o seu resultado.
- Agora defina prioridades distintas para elas e veja como fica o resultado.

Estácio | FIB

Interrompendo uma Thread

- Uma thread termina quando seu método run() retorna (encerra)
- Entretanto, quando uma thread está dormente, ela não pode verificar ativamente se deve terminar
  - Threads devem "dormir" de vez em quando para dar uma chance de trabalho a outras threads **(devemos evitar as chamadas "Threads Egoístas")**
- Para isso basta utilizar os métodos interrupt() e interrupted()

Interrompendo uma Thread

- void interrupt()
  - Envia um pedido de interrupção para uma thread;
  - Status de "interrompida" da thread é definido como true;
  - Se a thread foi bloqueada por uma chamada ao método sleep() ou wait(), uma InterruptedException será lançada
- static boolean interrupted()
  - Testa se a thread corrente (thread que está executando essa instrução) foi interrompida ou não
  - Reinicializa o status de "interrompida" da thread para false

Grupos de Threads

- Alguns programas possuem muitas threads, portanto, se torna interessante agrupá-las pela funcionalidade.
  - Ex: um browser carregando imagens em uma página web, pode ter uma solicitação para interromper esta operação. Neste caso ele interrompe o carregamento de todas as imagens de uma só vez.
- A linguagem Java permite que se construam grupos de threads para poder trabalhar simultaneamente com várias threads:
  - `String nomeGrupo = ...; //deve ser exclusivo`
  - `ThreadGroup g = new ThreadGroup(nomeGrupo);`

Grupos de Threads

- Para inserir uma thread no grupo, basta especificar o grupo no construtor da thread
  - `Thread t = new Thread(g, “nomeDoThread”);`
- Para saber se algumas threads de um grupo em particular ainda são passíveis de execução, use o método activeCount():
  - `if (g.activeCount()==0)`
  - `  // todas as threads no grupo g pararam`
- Para interromper todas as threads de um grupo, basta chamar interrupt() no grupo:
  - `g.interrupt();`

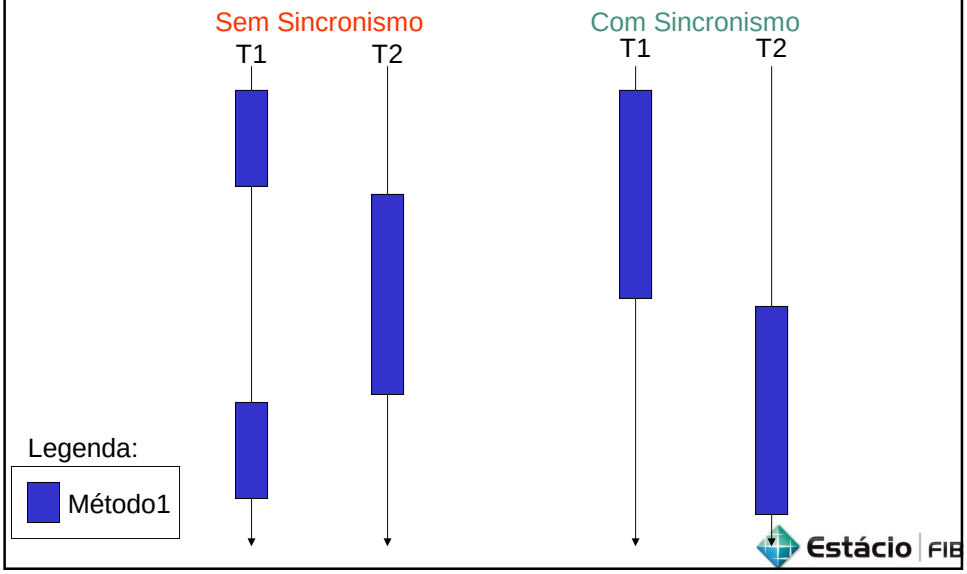
Sincronismo

- Na maioria dos aplicativos multithreads, duas ou mais threads precisam compartilhar o acesso aos mesmos objetos
- O que acontece se dois threads têm acesso ao mesmo objeto e cada um chama um método que modifica o estado do objeto?
  - As threads iniciam uma corrida pelo acesso aos objetos;
  - Dependendo da ordem de acesso, os objetos podem sair danificados.
- Tal situação é frequentemente chamada de **condição de corrida (race condition)**!!

Sincronismo

- Para evitar esse tipo de problema, é necessário **sincronizar** o acesso à um objeto compartilhado.
- Em Java, toda operação que não deve ser interrompida deve ser identificada como **synchronized**:
  - Exemplo de declaração: **public synchronized void transfer(...);**
- Algumas instruções **parecem**, mas **não são atômicas**, e por isso devem ser sincronizadas, ex: *vetor1[indice]* += *valor;*
  - Carrega **vetor1[indice]** em um registrador
  - Soma **valor**
  - Move o resultado de volta para **vetor[indice]**

Sincronismo x Não sincronismo



Bloqueio de Objetos

- Quando uma thread chama um método synchronized, o objeto se torna “bloqueado”
- Como outras threads tentarão acessar o objeto, periodicamente o scheduler (**escalonador**) ativará estas threads:
  - As threads reativadas irão verificar se o objeto ainda está bloqueado;
  - Se não estiver, ela fica sendo a próxima a obter acesso exclusivo ao objeto.

Bloqueio de Objetos

- Entretanto, existem algumas situações em que é necessário esperar por dados específicos durante a operação de um objeto bloqueado

- Em Java, para esperar dentro de um método synchronized, deve-se chamar o método wait():

```
public synchronized void transfer(int origem, int
destino, int valor){
  while (contas[origem] < valor)
    wait();
  // transfere fundos
}
```


Bloqueio de Objetos

- Quando wait() é chamado dentro de um método synchronized, a thread corrente é bloqueada e libera o bloqueio de objeto
  - A thread corrente é colocada numa **lista de espera**

- Para remover a thread da lista de espera, uma outra thread deve chamar o método **notify()/notifyAll**() no mesmo objeto que chamou o método wait():

Bloqueio de Objetos

```
– public synchronized void transfer(int origem, int
  destino, int valor){
–   ...
–   contas[origem]-= valor;
–   contas[destino]+= valor;
–   notifyAll();
– }
```

- **A regra geral é** chamar **notify()/notifyAll()** quando o estado de um objeto mudar de um modo que seja vantajoso para as threads que estão esperando.