Introdução ao uso de *Threads* em Java

Daniel de Angelis Cordeiro

Aula para a disciplina
MAC 0438 – Programação Concorrente

Introdução

- *Threads*, também chamados de processos leves, permitem múltiplas atividades dentro de um único processo;
- Um mesmo processo pode conter múltiplas *threads* que parecem executar ao mesmo tempo;
- Java é a primeira linguagem de programação a incluir o conceito de *threads* na própria linguagem.

Por que utilizar *Threads*?

Há várias razões para utilizarmos *threads.*

Entre elas:

- Maior desempenho em ambientes multiprocessados;
- Responsividade em interfaces gráficas;
- Simplificação na modelagem de algumas aplicações.

Criando *Threads*

Há duas formas de criarmos uma *thread* em Java:

- Ou usamos herança e criamos uma classe que estende a classe **Thread**;

- Ou criamos uma classe que implementa a interface **Runnable**.

Usando Herança

Usando herança, nossa classe deve sobrescrever o método **public void run()**:

```
class Tarefa1 extends Thread {
    public void run() {
        for(int i=0; i<1000; i++)
            System.out.println("Usando herança");
    }
}
```

Usando a Interface *Runnable*

A interface **Runnable** nos obriga a implementar o método **public void run()**:

```
class Tarefa2 implements Runnable {
    public void run() {
        for(int i=0; i<1000; i++)
            System.out.println("Usando
              Runnable");
    }
}
```

Exemplo de Uso

Para usar as classes **Tarefa1** e **Tarefa2** devemos fazer:

```
Thread threadComHeranca = new Tarefa1();
Thread threadComRunnable =
    new Thread(new Tarefa2());

threadComHeranca.start();
threadComRunnable.start();
```

Saída do Exemplo

A saída de nosso exemplo é mais ou menos essa:

Usando Runnable

Usando Herança

Usando Herança

Usando Runnable

Usando Herança

Usando Runnable

(...)

Estados de uma Thread

Uma *thread* pode estar em um dos seguintes estados:

- **criada**;
- **em execução**;
- **suspensa** ou
- **morta**.

Sincronismo entre *Threads*

- O uso de memória compartilhada entre as *threads* obriga o programador a sincronizar as ações de suas *threads*.
- Para isso, Java provê *monitores* ou *locks*. Imagine um *lock* como uma permissão para que apenas uma *thread* possa utilizar um recurso por vez.
- Cada objeto em Java possui um *lock* e ele deve ser obtido através do comando *synchronized*.

O comando *synchronized*

Os seguintes usos do comando **synchronized** são equivalentes:

```
synchronized public void teste()
{
    façaAlgo();
}
```

```
public void teste() {
    synchronized(this) {
        façaAlgo();
    }
}
```

Variáveis Voláteis

Variáveis voláteis possuem as seguintes características:

- operações atômicas;
- mudanças visíveis globalmente;
- são instáveis.

Exemplo de código:

```
class VolatileTeste {
    boolean flag;
    public void foo() {
        flag = false;
        if(flag) { (...) }
    }
}
```

wait(), notify() e notifyAll()

Estes métodos da classe **Object** implementam o conceito de monitores sem utilizar espera ativa. Ao invés disso, notificam as *threads* indicando se estas devem ser suspensas ou se devem voltar a ficar em execução.

O *lock* do objeto chamado pela *thread* para realizar as notificações será utilizado. Por isso, antes de chamar um dos três métodos, o *lock* deve ser obtido utilizando-se o comando **synchronized**.

wait(), notify() e notifyAll()

- **wait()** - suspende a *thread* que chamou o método até que outra *thread* a acorde ou até que o tempo especificado como argumento tenha passado;
- **notify()** - acorda, se existir, alguma *thread* que esteja esperando um evento neste objeto;
- **notifyAll()** - acorda todas as *threads* que estejam esperando neste objeto.

Mais Informações

Se mesmo depois desta *maravilhosa* aula você ainda tiver alguma dúvida, recomendo a leitura do texto que preparei sobre *threads* em Java (que está muito mais completo).

Para cada referência do texto coloquei um apontador para a página onde ela se encontra na internet. Assim você nem precisa procurar o livro na biblioteca para consultar.

:-)