PRÁTICA 03 – THREADS EM JAVA

Tales Bitelo Viegas <talesbv@ulbra.edu.br>

Universidade Luterana do Brasil (Ulbra) – Curso de Ciência da Computação – Câmpus Gravataí
Av. Itacolomi, 3.600 – Bairro São Vicente – CEP 94170-240 – Gravataí - RS

1 THREADS

Em um programa Java é possível definir diferentes sequências de execução independentes. A isto chamamos Threads.

Uma Thread é similar a um processo, no sentido em que corresponde a um conjunto de instruções que pode ser escalonado para execução em um dado processados. No entanto, as Threads definidas em um dado programa compartilham o mesmo espaço de endereçamento que o processo principal que lhes deu origem.

Existem duas formas de criar uma Thread em Java. A primeira consiste em criar uma **subclasse da classe Thread** e usar o método start() definido em Thread para iniciar a execução. O método start() invoca, por sua vez, o método run() a ser definido pelo programador, e que conterá a sequência de comandos da Thread instanciada.

Teste as classes abaixo:

```
public class MyThread extends Thread{

	public void run(){
		System.out.println("Olha quem está mandando um oi: " + getName());
	}
}

public class Teste {
	public static void main(String[] str) {
		MyThread Ta, Tb;
		Ta = new MyThread();
		Tb = new MyThread();
		Ta.start();
		Tb.start();
	}
}
```

Figura 1 – Classes de Teste de Threads

Este processo de criar uma Thread não pode ser usado se pretendermos que a classe que desejamos criar seja a subclasse de uma outra classe, pois Java não permite herança múltipla. Assim, a segunda forma de criar uma Thread é usar a interface **Runnable**. Em termos simples, "interface" é a definição de um conjunto de métodos que a classe ou classes que implementam esta interface terão de implementar.

No exemplo abaixo, a classe MyThread2 implementa a interface Runnable, implementando o método run(), que é o único método definido na interface. A diferença entre esta classe e a classe MyThread definida anteriormente é que a classe MyThread2 não herda os métodos da classe Thread. Assim, para criarmos uma nova Thread é necessário passar ao construtor desta Thread uma instância da classe que implementa a interface Runnable.

Implemente e estude o exemplo que se segue:

```
public class MyThread2 implements Runnable {
	public void run() {
		System.out.println("Olha quem está mandando um oi: " +
Thread.currentThread().getName());
	}
}

public class Teste2 {
	public static void main(String[] str) {
		MyThread2 T = new MyThread2();
		Thread Ta, Tb;
		Ta = new Thread(T);
		Tb = new Thread(T);
		Ta.start();
		Tb.start();
	}
}
```

Figura 2 – Classes de Teste de Threads com Runnable

2 "DAEMON" THREADS

Uma Thread "Daemon" é uma Thread geralmente usada para executar serviços em "background", que tem a particularidade de terminar automaticamente após todas as Threads "Não Daemon" terem terminado. Uma Thread transforma-se numa Thread Daemon através do método **setDaemon()**.

```
public class Normal extends Thread {
	public void run() {
		for (int i = 0; i < 5; i++) {
			try {
				sleep(500);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println("Eu sou uma Thread normal");
		}
		System.out.println("A Thread normal está encerrando");
	}
}

public class Daemon extends Thread {
	public Daemon() {
		setDaemon(true);
	}

	public void run() {
		for (int i = 0; i < 10; i++) {
			try {
				sleep(500);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println(" Eu sou a Thread Daemon");
		}
	}
}
```

Figura 3 – Threads Daemon

Depois de implementar estas duas classes, construa uma classe que teste as classes anteriores. Após observar o comportamento do programa, experimente comentar a linha onde o método setDaemon é invocado e observe a diferença de comportamento do programa.

3 GRUPOS DE THREADS

As Threads de um programa podem ser agrupadas, tornando possível enviar uma mensagem simultaneamente a um conjunto de Threads.

Considere as classes abaixo Execute e verifique o comportamento.

```
public class Normal extends Thread {
	public void run() {
		for (int i = 0; i < 5; i++) {
			try {
				sleep(500);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println("Eu sou uma Thread normal");
		}
		System.out.println("A Thread normal está encerrando");
	}
}

public class Daemon extends Thread {
	public Daemon() {
		setDaemon(true);
	}

	public void run() {
		for (int i = 0; i < 10; i++) {
			try {
				sleep(500);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println(" Eu sou a Thread Daemon");
		}
	}
}
```

Figura 4 – Grupos de Threads

Um grupo pode ser criado explicitamente, instanciando um objeto da classe ThreadGroup. Para adicionar uma Thread ao grupo criado, deverá criar um construtor da sua subclasse de Thread que inicialize o nome do grupo na superclasse Thread, usando o super() e passando por parâmetro o objeto ThreadGroup criado e o nome da Thread.

4 PRIORIDADE DE UMA THREAD

Java suporta 10 níveis de prioridades. A menor prioridade é definida por Thread.MIN_PRIORITY e a mais alta por Thread.MAX_PRIORITY. Podemos saber a prioridade de uma Thread através do método getPriority() e podemos modifica-la com o método setPriority(int).

5 SINCRONIZAÇÃO DE THREADS

A sincronização de Threads em Java é baseada no conceito do Monitor (de Hoare). Cada objeto Java tem associado um monitor (ou "lock") que pode ser ativado se a palavra-chave **syncronized** for usada na definição de pelo menos um método de instância:

public synchronized void metodoX();

O monitor da classe será ativado se um método de classe for declarado como synchronized:

public static synchronized void metodoY();

Se várias Threads invocarem um método declarado como **synchronized** ao mesmo tempo, o monitor associado ao objeto garantirá a execução desse método em exclusão mútua.

5.1 EXEMPLO

Suponha dois processos **p1** e **p2** que compartilham uma variável comum (variavelPart). Pretende-se construir um exemplo que ilustre a violação de uma seção crítica, sem usar qualquer tipo de mecanismo de sincronização.

Considere que o processo p1 possui duas variáveis locais, x e y, inicializados com valores simétricos, e que dentro de um ciclo infinito transfere a quantidade armazenada em variavelPart de x para y. O processo 2 vai, em cada iteração, incrementar a variável compartilhada.

Pretende-se que a condição x + y = 0 seja verdadeira durante toda a execução do programa. Quando, no processo p1, se detecta que a seção crítica foi violada (porque x + y != 0) o processo deve terminar e acabar o programa.

Supondo a estrutura que se segue para os processos p1 e p2, comece por criar duas classes que permitam criar os processos (Threads) p1 e p2. Estes dois processos deverão, por exemplo, compartilhar um valor do tipo array de inteiros com um elemento.

Processo 1	**Processo 2**

`x=M; y=-M;`
`while (true){`
`  // seção critica 1`
`  x-=variavelPart;`
`  y+=variavelPart;`
`  ...`
`  if(x+y != 0){`
`  print ''seção crítica violada'';`
`  break;`
`  }`
`}`
	`...`
	`while(true){`
	`  // seção crítica 2`
	`  variavelPart+=1;`
	`  ...`
	`}`

Construa uma classe de teste que, instanciando os processos p1 e p2, permita simular a violação da seção crítica.

Para que o processo p2 termine, após a violação da seção crítica, transforme-o em uma Thread Daemon.

Modifique o programa de maneira a garantir a execução de cada seção crítica em exclusão mútua.

5.2 PARA EXPLORAR

O exemplo abaixo ilustra a sincronização de um método de classe e de um método de objeto e instância.

Depois de o estudar, implemente-o e analise os resultados.

```
public class CriticaUm {
	public synchronized void method_A() {
		System.out.println(Thread.currentThread().getName() + " Método A");
		try {
			Thread.sleep((int) Math.round(Math.random() * 5000));
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.println(Thread.currentThread().getName() + " Saindo do Método A");
	}

	public static synchronized void method_B() {
		System.out.println(Thread.currentThread().getName() + " Método B");
		try {
			Thread.sleep((int) Math.round(Math.random() * 5000));
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.println(Thread.currentThread().getName() + " Saindo do Método B");
	}
}

public class Monitors extends Thread {
	CriticaUm c;

	public Monitors(String nomeobjeto) {
		Thread Thread_a, Thread_b;
		c = new CriticaUm();
		Thread_a = new Thread(this, nomeobjeto + ":Thread a");
		Thread_b = new Thread(this, nomeobjeto + ":Thread b");
		Thread_a.start();
		Thread_b.start();
	}

	public void run() {
		c.method_A();
		CriticaUm.method_B();
	}
}

public class Teste {
	public static void main(String args[]) {
		Monitors m1, m2;
		m1 = new Monitors("Objeto 1");
		m2 = new Monitors("Objeto 2");
	}
}
```

Figura 5 – Exemplo Synchronized