

→ Sincronização de Threads

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

```
public synchronized void metodoX();
```

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

```
public static synchronized void metodoY();
```

Se várias Threads invocarem um método declarado como **synchronized** em simultâneo o monitor associado ao objecto garantirá a execução desse método em exclusão mútua.

→ Monitores de classe e monitores de objecto

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

Exercício 1: - 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)  
        { ... }  
        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)  
    { ... }  
    System.out.println ( Thread.currentThread().getName() + " Saindo do Método B");  
}  
  
public class Monitors extends Thread{  
    CriticaUm C;  
  
    public Monitors(String nomeObjecto) {  
        Thread Thread_a, Thread_b;  
        C= new CriticaUm();  
        Thread_a = new Thread (this, nomeObjecto + ":Thread a");  
        Thread_b = new Thread (this, nomeObjecto + ":Thread b");  
  
        Thread_a.start();  
        Thread_b.start();  
    }  
  
    public void run (){  
        C.method_A();  
        C.method_B();  
    }  
}  
  
public class Teste {  
    public static void main (String args[]){  
        Monitors M1, M2;  
        M1=new Monitors ("Objecto 1");  
        M2=new Monitors ("Objecto 2");  
    }  
}
```

→ **Invocação recursiva de métodos sincronizados (código reentrante)**

Uma Thread que adquiriu um lock num objecto (ao executar um método sincronizado) pode invocar recursivamente o mesmo método. O lock é readquirido pela Thread.

Exercício 2: - Estude o exemplo abaixo.

```
public class Reentrant {  
    static int times =1;  
    public synchronized void myMethod() {  
        int i = times++;  
        System.out.println("myMethod has started " + i + " time(s)");  
        while (times <4)  
            myMethod();  
        System.out.println("myMethod has exited " + i + " time(s)");  
    }  
}  
  
public class Teste extends Thread {  
    Reentrant R;  
    public Teste (){  
        R= new Reentrant();  
        Thread T = new Thread(this);  
        T.start();  
    }  
  
    public void run(){  
        R.myMethod();  
    }  
  
    public static void main(String args[]) {  
        Teste T1 = new Teste();  
    }  
}
```

Exercício 3: - Suponha dois processos **p1** e **p2** que partilham uma variável comum, **variavelPart**. Pretende-se construir um exemplo que ilustre a violação de uma secção crítica, sem usar qualquer tipo de mecanismo de sincronização,

Considere que o processo p1 possui duas variáveis locais, x e y, inicializadas 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 a.

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

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

Processo 1

```
x = M; y = - M;
While (true){
  //secção crítica 1
  x = x - variavelPart;
  y = y + variavelPart;
  <parte restante 1>
  if (x+y != 0 ){
    print "Secção crítica violada"
    break;
  }
  ...
}
```

Processo 2

```
...
While (true){
  //secção crítica 2
  variavelPart =
    variavelPart +1;
  <parte restante 2>
}
...
```

- b) Construa uma classe de teste que, instanciando os processos p1 e p2, lhe permita simular a violação da secção crítica.
- c) Para que o processo p2 termine, após a violação da secção crítica, transforme-o numa Thread daemon.
- d) Pretende-se agora garantir a execução de cada secção crítica em exclusão mútua. Para isso, transforme a variável partilhada num objecto partilhado e sincronize o acesso ao objecto através da instrução synchronized.

→ **Multithreaded Servers**

Exercícios:

- 4 – Implemente o Servidor da data e hora do sistema estudado na aula teórica (“multithreaded server”).
- 5 – Modifique o programa anterior de modo a conseguir demonstrar experimentalmente que o servidor pode servir vários clientes em simultâneo.
- 6 – Construa uma aplicação cliente / servidor em que o servidor receba do cliente dois arrays de inteiros, calcule a sua soma e devolva o resultado ao cliente. O servidor deverá poder servir vários clientes simultaneamente. Para isso deverá possuir um conjunto de Threads definidas à partida (“pool de treads”) as quais estarão à espera de pedidos dos clientes.