→ Threads

⇨ "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)
    {...}
   System.out.println (" I' m the normal Thread");
  }
  System.out.println (" The normal Thread is exiting");
 }
}
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)
    { }
   System.out.println (" I'm a daemon Thread");
}  }
}
```

1 - Depois de implementar estas duas classes, construa uma classe em que teste ambas 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.

⇨ Grupos de Threads

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

2 - Considere as classes abaixo. Complete a classe Teste e teste-a.

```
public class MyThread extends Thread{
  public MyThread( String name) {
   super(name);
  }
  public void run (){
    while (true) {
      System.out.println ("Sou a " + this.getName());
      if ( isInterrupted() )
        break;
      yield();
    } }
}
public class Teste {
  public static void main (String[] arg){
    MyThread Ta, Tb, Tc;
    ThreadGroup this_group;
    this_group = Thread.currentThread().getThreadGroup();
    System.out.println("O nome do grupo é: " + this_group.getName());
    System.out.println("O nº de Threads activas no grupo é " + this_group.activeCount());

    Ta=new MyThread ("Thread A");
    Tb=new MyThread ("Thread B");
    Tc=new MyThread ("Thread C");

    // inicie a execução das Threads
   ...
    // obtenha o nome do grupo e o número de threads activas nesse grupo
   ...
  try
      {Thread.sleep (500);}
    catch (InterruptedException e)
      {...}

// Pode invocar um método em todas as Threads do grupo:
    this_group.interrupt();
  }
}
```

Um grupo pode ser criado explicitamente:

...
ThreadGroup Mygroup = new ThreadGorup (“ O meu grupo”)

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:

```
class MyThread extends Thread {
 public MyThread ( ThreadGroup tg, String name) {
   super(tg, name);
  }
...
```

3 - Teste a criação de dois grupos de threads num mesmo programa.

- ⇨ **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 *int getPriority()* e podemos modificá-la com o método *setPriority(int*).

4 - Verifique qual a prioridade por omissão de uma thread

5 – Construa um programa exemplo que lance 3 ou mais Threads, atribua prioridades diferentes às várias threads e estude o comportamento do programa.

→ 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.

6 - 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 partilhada.

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 **!=** 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 (Threads) 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;
  }//fim do if
  …
}// fim do While
```

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, 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) Modifique o programa de maneira a garantir a execução de cada secção crítica em exclusão mútua.

7 – Suponha que uma **sala de cinema** pretendia um pequeno programa que lhe permitisse gerir a venda de bilhetes em diferentes postos de venda. A sala tem uma lotação fixa, e a cada bilhete vendido é atribuído um número sequencial, por ordem de aquisição. Pretende-se poder:

- consultar o nome do filme em exibição;
- consultar o número de bilhetes disponíveis para venda;
- vender um bilhete (indicando ao utilizador o número correspondente ao bilhete em venda).

a)Construa uma classe, SalaCinema, que lhe permita realizar estas operações.

b)Para testar o programa começou-se por simular a sua execução concorrente, sendo cada posto de venda uma Thread que acede à classe anterior. Construa a classe que simula o posto de venda, PostoVenda, e uma classe de teste onde é criada a sala de cinema e os 3 postos de venda.

Para explorar:

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

Exercício - 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 objeto (ao executar um método sincronizado) pode invocar recursivamente o mesmo método. O lock é readquirido pela Thread.

Exercício - 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();
  }
}
```