Mecanismo de Threads em Java 2

Índice

MULTITHREADING : Múltiplos Processos

As *Threads* são convenientes para programar. Permitem concentrar numa tarefa sem que o programa tenha que se preocupar como é que vai interagir com outras tarefas.

Se for necessário ter de "trabalhar" duas tarefas em paralelo, simplesmente implementa-se uma *thread* para cada tarefa e iniciam-se essas *threads*. Há no entanto que ter alguns cuidados.

Observações:

As tarefas podem ter que ser sincronizadas.
Uma *thread* pode necessitar de um resultado que está a ser processado por outra *thread.*

Procedimentos para implementar *threads*:

1) Colocar o código da tarefa no método **run( )** da subclasse:

```
run( ){
        ... // tarefa a realizar
}
```

2) Construir um objecto da subclasse de **Thread**

```
minhaThread  novoProcesso = new Thread( ) ;
```

3) Chama-se o método **start( )** para iniciar a tarefa

```
novoProcesso.start( ) ;
```

Aviso:

Chamar o processo **run( )** é **INCORRECTO!**

Um exemplo:

Imprimir no écrã as mensagens: "Olá, LPG2!" e "Adeus, LPG2!", uma de cada por segundo e imprimir também o tempo para cada mensagem para se ver quando são impressas.

Definição da classe que é subcalasse de Thread:

```
public class minhaThread
```

Acções para a *thread*:

1. Imprimir a data
2. Imprimir a mensagem
3. Esperar um segundo

Obs.: *constrói-se um objecto Date para se poder imprimir data e horas*

Ficheiro minhaThread.java:

```
import java.util.Date;

public class minhaThread extends Thread{
      private static final int REPETICOES = 10;
      private static final int ATRASO=1000;
      private String mensagem;


      public minhaThread(String msg){
         mensagem = msg;
      }

      // o método run é herdado da  classe Thread e é re-escrito:
      public void run(){
         try{
            for(int i=1; i<= REPETICOES; i++){
               Date agora = new Date();
               System.out.println( agora + " >> Mensagem: " + mensagem );
               sleep(ATRASO);
            }
         }
         catch(InterruptedException ie){
            }
       }
}
```

Apesar de se poder testar o código neste mesmo ficheiro, fazê-lo seria contra a filosofia da programação orientada a objectos, e assim a *minhaThread* pode ser executada a partir de uma classe teste com o método estático **main(**).

Ficheiro TesteMinhaThread.java:

```
import java.util.Date;     // pacote que contém a classe Date

public class TesteMinhaThread{
   public static void main(String[ ] args) throws java.lang.InterruptedException{
      minhaThread mt1 = new minhaThread("Ola, LPG2!");
      minhaThread mt2 = new minhaThread("Adeus, LPG2!");
```

```
        // inicia a 1ª thread
        mt1.start() ;
        mt1.sleep(500);                    // espera ½ segundo...
       // inicia a 2ª thread
       mt2.start() ;
    }
}
```

Durante a classe de teste podem usar-se vários métodos dos quais se destacam os métodos **start( )** que tem a função de executar o código que se escreveu no método **run( )** e o método **sleep( )** que vai suspender a *thread* por um tempo parametrizado em milisegundos.
Eis um *output* da execução desta aplicação:

```
Sun May 23 16:34:18 BST 2004 >> Mensagem: Ola, LPG2!
Sun May 23 16:34:19 BST 2004 >> Mensagem: Adeus, LPG2!
Sun May 23 16:34:19 BST 2004 >> Mensagem: Ola, LPG2!
Sun May 23 16:34:20 BST 2004 >> Mensagem: Adeus, LPG2!
Sun May 23 16:34:20 BST 2004 >> Mensagem: Ola, LPG2!
Sun May 23 16:34:21 BST 2004 >> Mensagem: Adeus, LPG2!
Sun May 23 16:34:21 BST 2004 >> Mensagem: Ola, LPG2!
Sun May 23 16:34:22 BST 2004 >> Mensagem: Adeus, LPG2!
Sun May 23 16:34:22 BST 2004 >> Mensagem: Ola, LPG2!
Sun May 23 16:34:23 BST 2004 >> Mensagem: Adeus, LPG2!
Sun May 23 16:34:23 BST 2004 >> Mensagem: Ola, LPG2!
Sun May 23 16:34:24 BST 2004 >> Mensagem: Adeus, LPG2!
Sun May 23 16:34:24 BST 2004 >> Mensagem: Ola, LPG2!
Sun May 23 16:34:25 BST 2004 >> Mensagem: Adeus, LPG2!
Sun May 23 16:34:25 BST 2004 >> Mensagem: Ola, LPG2!
Sun May 23 16:34:26 BST 2004 >> Mensagem: Adeus, LPG2!
Sun May 23 16:34:26 BST 2004 >> Mensagem: Ola, LPG2!
Sun May 23 16:34:27 BST 2004 >> Mensagem: Adeus, LPG2!
Sun May 23 16:34:27 BST 2004 >> Mensagem: Ola, LPG2!
Sun May 23 16:34:28 BST 2004 >> Mensagem: Adeus, LPG2!
```

Gestor de Threads (Scheduler)

A Java Virtual Machine disponibiliza um gestor de *threads*. Ao ser construído código com o mecanismo de *threads* estas são colocadas em agenda. Esta, não dá quaisquer garantias sobre a ordem em que várias *threads* vão ser executadas.

Cada *thread* corre por um pequeno espaço de tempo, chamada de *"time sli ce" – fatia de tempo*, então o gestor de *threads* atira outra qualquer escolhendo uma *thread* executável à disposição.

Uma *thread* está pronta a ser executada se não estiver em modo **sleep( )** ou bloqueada de outra forma.

Resumo: As várias *threads* depois de iniciadas pelo método **start( )** vão ser executadas aleatoriamente.

Gestor de Threads (Scheduler) : Prioridades

Podem atribuir-se diferentes prioridades na implementação do código utilizando o método **setPriority(** *int prioridade* **)**, com um parâmetro do tipo primitivo inteiro compreendido no intervalo 0 – 10. Por "defeito" a prioridade é 5.

valor	atributo estático
0	Thread.MIN_PRIORITY
...	…
5	Thread.NORM_PRIORITY
...	…
10	Thread.MAX_PRIORITY

Caso este método receba um inteiro fora deste intervalo é lançada uma excepção ***IllegalArgumentException*** e lança uma ***SecurityException*** se uma *thread* em execução não puder alterar o valor desta.

Sempre que o gestro de threads precisa de decidir qual a *thread* a executar, escolhe a que tiver o valor de prioridade mais elevado. Se existir mais do que uma *thread* com a

mesma prioridade, qualquer umas deles será escolhida sem qualquer outro critério de selecção.

Se uma *thread* com prioridade elevada "a corda" enquanto está a ser executada outra com menos prioridade o gestor suspende a sua execução e activa a thread com maior prioridade.

Tendo em conta programas actuais com objectos actuais as prioridades parecem ser muito úteis, mas tendo em conta uma das principais características da programação orientada a objectos: *a hereditariedade*, as prioridades podem tornar-se perigosas na re-utilizição de código. O mecanismo de threads fornece outra ferramenta que desarma esse perigo eventual: a sincronização.

Terminar Threads

O método normal de terminar threads é quando o método **run( )** retorna. Por vezes pode haver necessidade de terminar uma thread em execução. Por exemplo, se várias threads estiverm a tentar resolver o mesmo problema e uma delas encontrar a solução, torna-se necessário notificar as outras para terminarem a sua execução.

Usa-se o método **interrupt( )**. Primeiro o método **checkAccess( )** é executado e depois a thread é terminada. Para verificar se o estado da thread foi alterado ou não pode usar-se o método **isInterrupted( )**. No exemplo de criação de data e impressão de uma mensagem podia ter-se escrito da seguinte maneira:

```
public void run(){
        try{
           for( int i=1;
                i <= REPETICOES && ! isInterrupted() ;
                 i++){

                      Date agora = new Date();
                      System.out.println( agora + " >> Mensagem: " + mensagem );
                      sleep(ATRASO);
           }
        }
        catch(InterruptedException ie){
           }
      }
```

Grupos de Threads

Num programa com muitas threads, estas podem ser colocadas num grupo. Escolhe-se um nome:

String nome = "me uGrupo" ;

ThreadGroup ***grupo*** *= new ThreadGroup(nome) ;*

A grande vantagem é que se for necessário as threads podem ser geridas em simultâneo. No caso de ser necessário utilizar o método **interrupt( )** para todas as threads em execução basta escrever:

grupo.interrupt( ) ;

Um exemplo prático é a situação em que várias threads carregam várias partes de uma página web e o utilizador decide que já não quer ver essa página, o programa pode então terminar todas as threads em simultâneo.

Observações:

Várias threads podem ser sincronizadas.
Duas ou mais threads podem aceder ao mesmo objecto.

O somatório destas duas características pode levar a resultados inesperados.

O acesso partilhado a um objecto cria um problema que é conhecido como: *Race-Condition*.

As threads, cada qual no seu tempo de execução para executar a sua tarefa, manipulam um campo partilhado e o resultado depende de qual thread *vence* a competição.

Se tivermos uma thread que tenha a função num determinado período de tempo de incrementar um objecto de classe NumEncomendas
(número de encomendas)

```
for (int i=0; i < REPETIR; i++ ){
        enc.aumentaEncomenda(10);
        System.out.print( (i+1) + ": Efectadas + " + quantidade + " Encomendas. ");
         System.out.println("Total de encomendas: " + enc.getNumEncomendas() );

        sleep(ATRASO);  // atrasa 1/2 segundos
    }
```

e que tenhamos outra thread que tenha como função num determinado período de tempo de anular o número de encomendas do mesmo objecto pode levar a resultados enganadores.

```
for(int i=0; i < REPETIR; i++ ){
        enc.anulaEncomenda(10);
        System.out.print("Anuladas " + quantidade + " Encomendas.");
        System.out.println("Total de encomendas: " +enc.getNumEncomendas() );

        sleep(ATRASO );  // atrasa 1/2 segundos
    }
```

Se tudo correr bem o *output* esperado será de que em 10 iterações (atributo do tipo primitivo final e estático 'REPETIR') em que haja 10 encomendas e 10 anulações o resultado de **enc.getNumEncomendas( )** em que **enc** é um objecto do tipo parametrizado **NumEncomenda** será de zero encomendas.

Por exemplo ao invocar numa classe teste os seguintes métodos:

```
public class TestNumEnc{

   public static void main(String[] args){
      NumEncomendas enc = new NumEncomendas();

      ThreadEfectuarEnc thread1 = new ThreadEfectuarEnc( enc, 10 );
      ThreadAnularEnc thread2 = new ThreadAnularEnc( enc, 10 );

      thread1.start() ;
      thread2.start() ;

   }
}
```

O output esperado será:

```
1: Efectadas + 10 Encomendas. Total de encomendas: 10
Anuladas 10 Encomendas.Total de encomendas: 0
2: Efectadas + 10 Encomendas. Total de encomendas: 10
Anuladas 10 Encomendas.Total de encomendas: 0
3: Efectadas + 10 Encomendas. Total de encomendas: 10
Anuladas 10 Encomendas.Total de encomendas: 0
4: Efectadas + 10 Encomendas. Total de encomendas: 10
Anuladas 10 Encomendas.Total de encomendas: 0
5: Efectadas + 10 Encomendas. Total de encomendas: 10
Anuladas 10 Encomendas.Total de encomendas: 0
6: Efectadas + 10 Encomendas. Anuladas 10 Encomendas.Total de encomendas: 0
Total de encomendas: 0
7: Efectadas + 10 Encomendas. Total de encomendas: 10
Anuladas 10 Encomendas.Total de encomendas: 0
8: Efectadas + 10 Encomendas. Total de encomendas: 10
Anuladas 10 Encomendas.Total de encomendas: 0
9: Efectadas + 10 Encomendas. Total de encomendas: 10
Anuladas 10 Encomendas.Total de encomendas: 0
10: Efectadas + 10 Encomendas. Total de encomendas: 10
Anuladas 10 Encomendas.Total de encomendas: 0
```

Mas se por qualquer motivo (ou até porque pode haver mais threads em execução) e a ordem das threads for trocada, a imprevisibilidade do resultado dispara.

Sincronização de Acesso um Objecto

Para resolver este tipo de problemas uma thread tem que ter a capacidade de fechar/trancar um objecto temporariamente. Assim, enquanto uma thread tiver um objecto trancado, nenhuma outra thread deve ser capaz de alterar o estado de um objecto.

Usam-se então os métodos sincronizados para bloquear os objectos. Basta marcar todos os métodos que sejam susceptíveis de conter código já afectado de threads com a palavra reservada *synchronized*.

```
public class NumEncomendas {
        public synchronized void anular( ) {
                …
        }
}
```

Cada objecto tem uma fechadura. Por defeito está “destrancada”. Uma thread pode “trancar” o objecto ao utilizar um método sincronizado, e este fica trancado até a thread abandonar o método.

Quando outra thread invoca um método sincronizado tem que aguardar até que a outra thread abandone o método.

Mais do que uma thread podem aguardar o “destrancar” de um método sincronizado. A thread que a seguir tem o direito de invocar e trancar o mesmo método é escolhida pelo gestor de threads de uma forma aleatória.

Apêndice A:

Ficha Técnica

Bibliografia Consultada:

Big Java, *[Cay Horstmann]*
Sebenta de LPG2 *[Fernando Mouta]*

Sistema(s) Operativo(s):

O código fonte dos exemplos apresentados foram escritos, compilados e executados em Suse9.0 Linux e em Windows XP Home Edition sem qualquer problema.

Interface Gráfico:

O ambiente de edição e compilação do código fonte foi o NetBeans IDE 3.5.1 da Sun Microsystems.

Máquinas:

Os programas foram executados com sucesso em duas máquinas:

i. Pentium III 990 MHz com 512 MB de RAM
ii. Pentium IV 2.40 GHz com 448 MB de RAM disponíveis

Java Virtual Machine:

j2sdk1.4.2

Aluno:

Paulo Jorge Morais Costa
n.º 1000334, turma 2AN
tlm: +351 938 312 943
tlf: +351 229 955 048
email: i000334@dei.isep.ipp.pt
web: http://www.dei.isep.ipp.pt/~i000334

Apêndice B:

Código - Fonte

```
// minhaThread.java

import java.util.Date;

public class minhaThread extends Thread{
        private static final int REPETICOES = 10;
        private static final int ATRASO=1000;
        private String mensagem;


        public minhaThread(String msg){
           mensagem = msg;
        }

        public void run(){
           try{
              for(int i=1; i<= REPETICOES; i++){
                 Date agora = new Date();
                 System.out.println( agora + " >> Mensagem: " + mensagem );
                 sleep(ATRASO);
              }
           }
           catch(InterruptedException ie){
              }


        }
}

// TesteMinhaThread.java

import java.util.Date;

public class TesteMinhaThread{
    public static void main(String[] args) throws java.lang.InterruptedException{
        minhaThread mt1 = new minhaThread("Ola, LPG2!");
        minhaThread mt2 = new minhaThread("Adeus, LPG2!");

        mt1.start() ;
```

```
        mt1.sleep(500);
        mt2.start() ;
    }
}
// ThreadEfectuarEnc.java

public class ThreadEfectuarEnc extends Thread{
    private static final int REPETIR=10;  // número de iteracoes a efectuar
    private static final int ATRASO=500; // valor da pausa entre processo; em
milisegundos
    protected NumEncomendas enc ;
    protected int quantidade ;

    /**
        Construtor de uma thread que serve para aumentar o número de
encomendas
        @param nE o objecto NumEncomendas que vai ser aumentado
        @param qtd a quantidade a incrementar
     */
    public ThreadEfectuarEnc( NumEncomendas nE, int qtd ){
        enc = nE ;
        quantidade = qtd ;
    }

    /**
        Método herdado da classe Thread e que tem que ser re-escrito
     */
    public void run(){
        try{
            for(int i=0;
                    i <= REPETIR && ! isInterrupted();
                     i++ ){
                            enc.aumentaEncomenda(10);
                            System.out.print( (i+1) + ": Efectadas + " + quantidade + "
                    Encomendas. ");
                            System.out.println("Total de encomendas: " +
                            enc.getNumEncomendas() );

                sleep(ATRASO);  // atrasa 1/2 segundos
            }
        }catch(InterruptedException ie){
        }
    }
}
```

// ThreadAnularEnc.java

```
public class ThreadAnularEnc extends Thread{
   private static final int REPETIR=10;  // número de iteracoes a efectuar
   private static final int ATRASO=2000; // valor da pausa entre processo; em
milisegundos
   protected NumEncomendas enc ;
   protected int quantidade ;

   /**
      Construtor de uma thread que serve para reduzir o número de
encomendas
      @param nE o objecto NumEncomendas que vai ser reduzido
      @param qtd a quantidade a anular
    */
   public ThreadAnularEnc( NumEncomendas nE, int qtd ){
      enc = nE ;
      quantidade = qtd ;
   }

   /**
      Método herdado da classe Thread e que tem que ser re-escrito
    */
   public void run(){
      try{
         for(int i=0;
               i <= REPETIR && ! isInterrupted();
                  i++ ){
            enc.anulaEncomenda(10);
            System.out.print("Anuladas " + quantidade + " Encomendas.");
            System.out.println("Total de encomendas: "
+enc.getNumEncomendas() );

            sleep(ATRASO);  // atrasa 1/2 segundos
         }
      }catch(InterruptedException ie){
      }
   }
}
```

```
// NumEncomendas.java

public class NumEncomendas{

   /**
      metodo que vai aumentar/actualizar a quantidade existente de
encomendas.
      @param quantidade a acrescentar ao número de encomendas que já
existem
    */
   public void aumentaEncomenda(int nova){
      int enc = nova ;
      numEnc += enc ;
   }

   /**
      metodo que vai retirar/actualizar a quantidade existente de encomendas.
      @param número de anulaçoes a efectuar
    */
   public void anulaEncomenda(int tira){
      int enc = tira;
      numEnc -= enc;
   }

   /**
      metodo que consulta o total de encomendas efectuadas.
      @return número total de encomendas
    */
   public int getNumEncomendas(){  return numEnc ; }

   private int numEnc ;  // número total de encomendas
}



// TestNumEnc.java

public class TestNumEnc{

   public static void main(String[] args){
      NumEncomendas enc = new NumEncomendas();

      ThreadEfectuarEnc thread1 = new ThreadEfectuarEnc( enc, 10 );
      ThreadAnularEnc thread2 = new ThreadAnularEnc( enc, 10 );

      thread1.start() ;
```

```
        thread2.start() ;

    }
}
```

Apêndice C:

Métodos não estáticos da Classe Thread

Métodos não estáticos que vão ser herdados pelas subclasses de Thread:

- **clone( )**
- **countStackFrames( )**
- **destroy( )**
- **equals(Object)**
- **finalize( )**
- **getContextClassLoader( )**
- **hashCode( )**
- **interrupt( )**
- **isInterrupted( )**
- **run( )**
- **setContextClassLoader( )**
- **start( )**
- **toString( )**

Apêndice D:

Disquete

Disquete com apresentação de slides do trabalho teórico

Threads.pps