Programação Concorrente em java - Exercícios Práticos

Abril 2004

1. Introdução

As *threads* correspondem a linhas de controlo independentes no âmbito de um mesmo processo. No caso da linguagem JAVA, é precisamente o conceito de *thread* que é utilizado como modelo para a programação concorrente, permitindo que uma aplicação em execução numa máquina virtual Java possa ter várias linhas de execução concorrentes em que cada uma corresponde a uma *threa*d. Esta ficha explora a utilização de threads em JAVA, bem como, de uma forma geral, os conceitos básicos de programação concorrente em JAVA. Como referências de apoio à realização desta ficha deverão ser utilizados os seguintes recursos:

Documentação do JDK em **http://marte.dsi.uminho.pt/javadocs/api/index.html**

Tutorial de Java Threads em
http://java.sun.com/docs/books/tutorial/essential/threads/index.html

2. Threads em Java

O suporte para *threads* na linguagem JAVA é realizado através da classe `java.lang.Thread` cujos métodos definem o API disponível para a gestão de *threads*. Entre as operações disponíveis incluem-se métodos para iniciar, executar, suspender e gerir a prioridade de uma *thread*. O código que define o que uma *thread* vai fazer é o que estiver no método `run`. A classe `Thread` é uma implementação genérica de uma *thread* com um método `run` vazio, cabendo ao programador definir esse código para uma *thread* em particular.

O ciclo de vida de uma *thread* começa com a criação do respectivo objecto, continua com a execução do método `run` (iniciada através da invocação do método `start`), e irá terminar quando terminar a execução do método `run`.

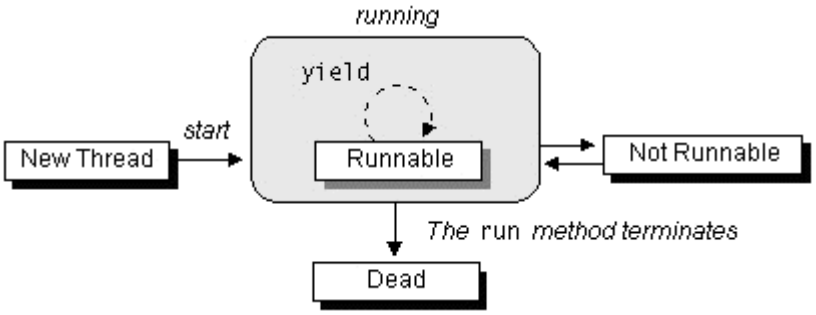

Figure 1 Ciclo de vida de uma *thread*

Umas das técnicas para criar uma nova *thread* de execução numa aplicação JAVA é criar uma subclasse de `Thread` e re-escrever o método `run`. A nova classe deverá redefinir o método `run` da classe `Thread` de forma a corresponder ao código que se pretende que a nova *thread* execute.

Exemplo 1

```
class PrimeThread extends Thread {

long minPrime;
PrimeThread(long minPrime) {
this.minPrime = minPrime;
}

public void run() {
// código a executar pela thread
. . .
}
}
```

A classe que pretendia criar a nova thread teria de executar o seguinte código:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

A outra forma de criar uma Thread é declarar uma classe que implementa o interface `Runnable` e que implementa o método `run`.

Exemplo 2

```
class PrimeRun implements Runnable {
long minPrime;
PrimeRun(long minPrime) {
this.minPrime = minPrime;
}

public void run() {
// código a executar pela thread
. . .
}
}
```

A classe que pretendia criar a nova thread teria de executar o seguinte código:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

3. Aplicações concorrentes em JAVA

O cenário que serve de base para esta ficha prática é o de um depósito de caixas. Um produtor vai ao depósito armazenar as caixas que vai produzindo. Um consumidor vai ao depósito para levantar essas mesmas caixas. O programa apresentado em baixo corresponde ao código do Depósito, permitindo a retirada e a colocação de elementos através dos métodos *retirar* e *colocar* respectivamente.

Programa 1 (Deposito.java)

```
public class Deposito {

private int items=0;
private final int capacidade=10;

public int retirar() {
if (items>0) {
items--;
System.out.println("Caixa retirada: Sobram "+items+" caixas");
return 1; }
return 0;
}

public int colocar () {
if (items<capacidade) {
items++;
System.out.println("Caixa armazenada: Passaram a ser "+items+"
caixas");
return 1; }
return 0;
}

public static void main(String[] args) {
Deposito dep = new Deposito();
Produtor p = new Produtor(d, 2);
Consumidor c = new Consumidor(d, 1);

//arrancar o produtor
//...
//arrancar o consumidor
//...

System.out.println("Execucao do main da classe Deposito terminada");
}
```

T1 >> Caso ainda não esteja, configure o seu ambiente de trabalho de forma a poder utilizar as ferramenta do Java SDK.
T2 >> Teste o ambiente de trabalho gerando o ficheiro `Deposito.class` e executando-o de seguida.
T3 >> Crie uma classe `Produtor` que funcione como uma *thread* independente e que vai invocando o método `colocar` da classe `depósito` de forma a acrescentar caixas ao depósito. A classe `Produtor` deve receber no construtor uma referência para o objecto `dep` onde os métodos vão ser invocados e um inteiro correspondente ao tempo em segundos entre produções de caixas. Defina a classe `Produtor` como sendo uma classe que implementa o método `Runnable`.
T4 >> Crie uma classe `Consumidor` que funcione como uma *thread* independente e que vai invocando o método retirar da classe `depósito` de forma a retirar caixas do depósito. A classe `Consumidor` deve receber no construtor uma referência para o objecto dep onde os métodos vão ser invocados e um inteiro correspondente ao tempo em segundos entre consumos de caixas. Defina a classe `Consumidor` como sendo uma sub-classe da classe `Thread`.
T5 >> Indique que motivos poderão levar a que se opte por realizar uma *thread* através da

implementação do interface `Runnable` ou como sub-classe de `Thread`.
T6 >> Ponha o sistema a funcionar e experimente algumas variantes, como por exemplo:

- Adicione à classe `consumidor` mensagens que permitam identificar o que cada objecto Consumidor está a fazer em cada momento, e em particular se está bloqueado à espera que existam caixas.
- Altere o número de consumidores ou de produtores e os tempos médios entre produções e consumos.

T7 >> Usando tempos de consumo e produção adequados, bem como alguns métodos `Thread.sleep` tente forçar a ocorrência de *race-conditions*.

4. Sincronização de threads em JAVA

A existência de *Threads* concorrentes levanta a necessidade de sincronização. Em JAVA, cada objecto tem associado um monitor através do qual é possível garantir o acesso exclusivo às secções críticas desse mesmo objecto. O controlo de acesso às secções críticas de um objecto é feito de forma automática pelo sistema de run-time do JAVA . O que o programador precisa de fazer é apenas assinalar as secções críticas usando para tal a primitiva `synchronized`. Um bloco de código sincronizado é uma região de código que apenas pode ser executado por uma thread de cada vez. Pode-se declarar um método como sendo synchronized:

```
synchronized int pull ()
{... }
```

Ou pode-se declarar apenas uma parte de um método como sendo synchronized:

```
synchronized (this)
{... }
```

T8 >> Analise o código das classes do exercício anterior e identifique potenciais problemas de concorrência na actual implementação da classe `Deposito`.
T9 >> Utilizando a primitiva `synchronized`, altere a implementação do sistema por forma a evitar a ocorrência de *race-conditions*.

4.1. Coordenação de threads em JAVA

Para além de evitar *race-conditions* também é importante permitir a coordenação entre *Threads*. Em JAVA esta coordenação faz-se usando os seguintes métodos da classe *`Object`*: *`wait, notify, e notifyAll`*. No seu conjunto estes métodos permitem às *threads* bloquear à espera de determinada condição ou notificar outras *threads* que se encontrem bloqueadas de que a condição pela qual estão à espera pode já se ter realizado.

T10 >> Identifique as razões pelas quais a utilização da primitiva `synchronized` poderá não ser suficiente para garantir um funcionamento adequada deste sistema.
T11 >> Consulte a informação referente aos seguintes métodos: `wait`, `notify`, e `notifyAll`.
T.12 >> Utilizando os métodos acima referido, altere a implementação do sistema por forma a permitir uma coordenação adequada entre os objectos.

5. Dependência de estado e guardas de métodos

A invocação de um método num objecto pode depender de condições associadas ao estado desse objecto. As acções dependentes do estado são métodos cuja execução pode não ser possível num determinado momento mas cuja viabilidade futura depende de algum evento externo que poderá ocorrer a qualquer momento. O método `retirar` na classe `Deposito` é um

exemplo de um método cuja invocação vai depender do estado do objecto no qual é invocado. Quando um objecto `Consumidor` tenta retirar uma caixa do depósito mas não existem quaisquer caixas disponíveis o método não pode prosseguir imediatamente. Contudo, é possível que a qualquer momento um `Produtor` coloque no depósito uma nova caixa e então o método já possa ser invocado.
Qual a atitude a tomar quando o método é invocado mas a pré-condição necessária à sua execução (por exemplo existirem caixas no depósito) não se verifica? Para decidir o que fazer nesses casos podem ser seguidas essencialmente duas abordagens:

- Uma abordagem optimista (liveness first) é assumir que mais tarde ou mais cedo a pré-condição há-de ser verdadeira e por isso espera-se.
- Uma abordagem conservadora (safety first) admite que a pré-condição pode nunca vir a ser verdadeira (pelo menos em tempo útil) e por isso se num determinado momento não é possível executar o método então mais vale devolver uma indicação de erro.
- Uma abordagem intermédia consiste em esperar mas definir um tempo limite.

Estas várias abordagens correspondem então às seguintes políticas:

- **Balking** Caso a invocação do método não seja possível a sua execução é recusada

Situações em que é mais indicado:

- **Guarded Suspension(guarded waits)** Execução é suspensa até que a pré-condição seja verdadeira. Implica que outras threads alterem o estado do objecto!
- **Timed Waits** Execução é suspensa até que a pré-condição seja verdadeira ou decorra um tempo máximo definido.

Altere a classe `Deposito` por forma a suportar três implementações diferentes do método *`retirar`*, cada uma delas correspondendo a cada uma das três políticas apresentadas em cima.

T.13 >> Implemente o método `retirar_balking()` que utiliza a estratégia de falhar em caso de impossibilidade de continuação.
T.14 >> Implemente o método `retirar_guarded_suspension()` que utiliza a estratégia de bloquear até que seja possível continuar.
T.15 >> Implemente o método `retirar_timed_wait(long milliseconds)` que utiliza a estratégia de bloquear até que seja possível continuar, mas que falha se ao fim do tempo indicado como parâmetro ainda não tiver sido possível continuar
T.16 >> Acrescente ao construtor da classe `Consumidor` um argumento que indique qual dos métodos deverá ser utilizado pelo objecto. Ponha cada um dos três consumidores a utilizar um método diferente para aceder à classe Deposito e analise qual deles é mais bem sucedido.

6. Questões finais

T.17 >> Algumas classes Java, como por exemplo a classe vector, dizem-se *threadsafe* ou *synchronize*d. Descreva o significado disto e as implicações que isso tem para os programadores que pretendem usar essas classes.

T.18 >> Imagine que se pretendia suportar em Java um mecanismo de semáforos. Como é que se poderia definir uma classe **semáforo** que fosse capaz de suportar a funcionalidade dos semáforos?