Programação Orientada e Objetos II

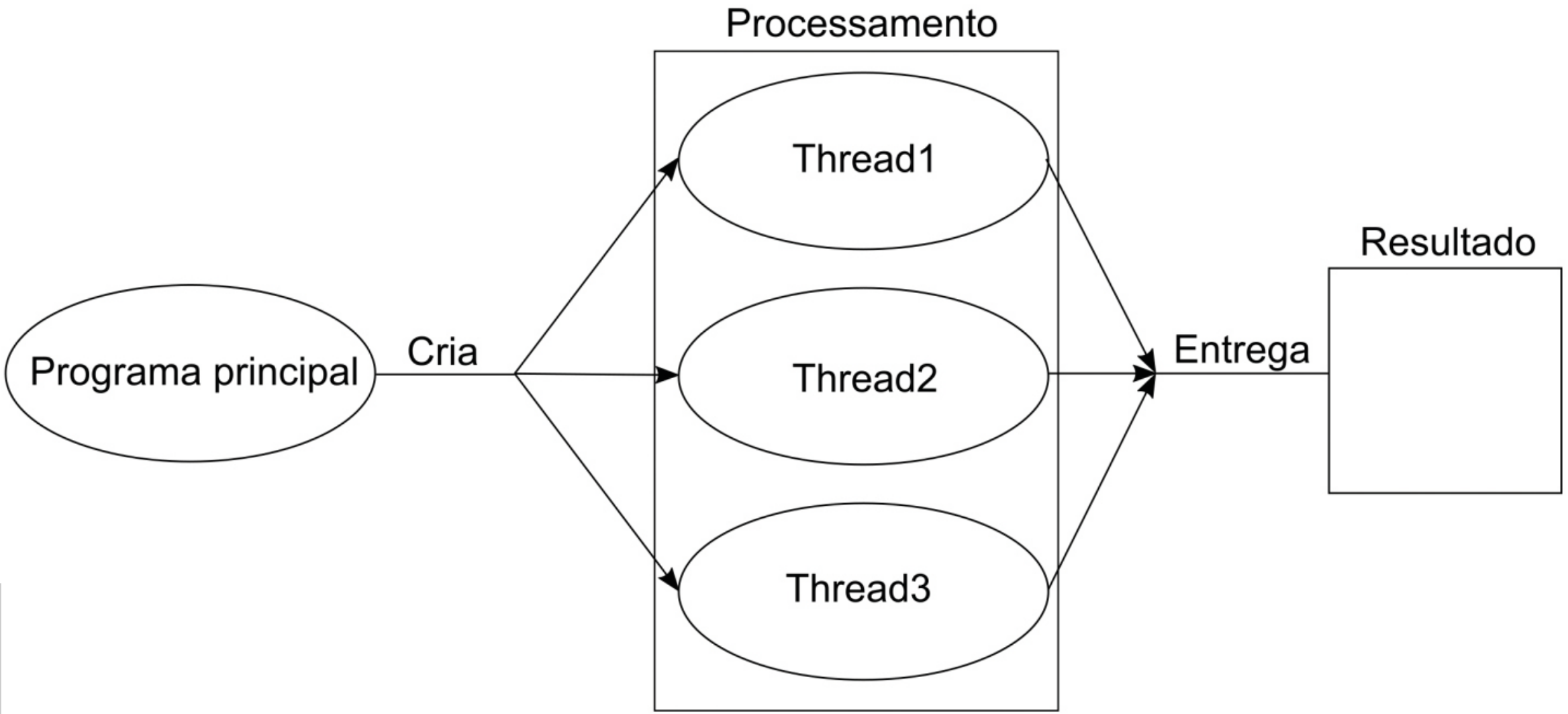
Anhanguera


AVALIE
SUA PROFISSÃO
QUANDO APARECER EM SEU PORTAL UMA AVALIAÇÃO SOBRE SEU CURSO, RESPONDA:
NOTAS
9 OU 10
SIGNIFICA QUE VOCÊ INDICA
NOTAS
7 OU 8
SIGNIFICA QUE VOCÊ NÃO INDICA
Anhanguera

Anhanguera

Programação em Java utilizando elementos para sincronização em Java

Existem diversas formas de tratar as exceções em um código orientado a objetos e paralelo, mas há problemas que podem ocorrer em sistemas concorrentes que não geram erros que são tratados pelos mecanismos do próprio Java. A utilização de threads em um código é essencial para que um software execute mais de uma tarefa ao mesmo tempo e, com isso, propicie mais interação com o usuário e use o hardware de maneira mais completa, entre outros cenários (MANZANO, 2014). Todavia, o uso da programação paralela pode gerar problemas decorrentes da execução de duas ou mais tarefas simultâneas. Para ilustrar esses cenários, podemos pensar em uma classe básica em que quantidades de elementos são somadas ou subtraídas. Nessa situação, imagine que diversas threads utilizam a mesma instância dessa classe para centralizar essas estatísticas. Esse tipo de cenário em programação é bem comum, e a Figura 2.3 apresenta um caso em que um programa principal cria três threads para fazer um processamento e, ao final, todas elas escrevem seus dados em apenas um elemento central.


Processamento
Thread1
Thread2
Thread3
Programa principal
Cria
Entrega
Resultado

O Quadro 2.18 apresenta um exemplo de classe para centralizar dados vindos de diversas threads. Contador representa o Resultado na Figura 2.3. Como é possível observar, a classe não possui nenhum recurso especial em termos de programação, mas em termos de funcionamento sim, pois como os dados devem ser centralizados usando-se várias threads, todas elas utilizam a mesma instância da classe Contador.

```
package U2S3;

public class Contador {
	private int quantidadeAlunosCurso;

	public void incrementa()
	{
	  quantidadeAlunosCurso++;
	}

	public void decrementa()
	{
	  quantidadeAlunosCurso--;
	}
}
```

No método incrementa(), o valor do quantidadeAlunosCurso é incrementado com operador ++, dessa forma, a operação consiste em:

- Recuperar o valor de quantidadeAlunosCurso.
- Incrementar o valor de quantidadeAlunosCurso em 1.
- Armazenar o valor alterado de quantidadeAlunosCurso.

O método decrementa() segue a mesma regra, apenas fazendo um decremento no valor quantidadeAlunosCurso. Em um cenário de apenas uma linha de execução não há problemas, porém, ao pensar que duas ou mais threads utilizarão a mesma referência dessa classe, podemos ter um problema de interferência de threads (DEITEL; DEITEL, 2016). Veja o que pode ocorrer quando duas threads fazem o acesso a um mesmo método na mesma referência:

- Thread 1: chama o incrementa().
- Thread 2: chama o decrementa().
- Thread 1: recupera o valor de quantidadeAlunosCurso.
- Thread 2: recupera o valor de quantidadeAlunosCurso.
- Thread 1: incrementa o valor de quantidadeAlunosCurso em 1, agora quantidadeAlunosCurso é 1.
- Thread 2: decrementa o valor de quantidadeAlunosCurso em 1, agora quantidadeAlunosCurso é -1.
- Thread1: armazenaovaloralteradode quantidadeAlunosCurso, agora quantidadeAlunosCurso é 1.
- Thread 2: armazena o valor alterado de quantidadeAlunosCurso, agora quantidadeAlunosCurso é -1.

Repare que o processamento que a thread 1 realizou foi sobrescrito e o valor do quantidadeAlunosCurso 1 está errado, pois as duas threads fizeram o acesso ao mesmo tempo. Dessa forma, são necessárias estratégias para que a própria linguagem de programação e o sistema operacional garantam que certos métodos sejam acessados por apenas uma thread por vez. Para isso, existe a palavra reservada synchronized, utilizada no método que vai garantir que cada thread acesse de forma única, a fim de evitar inconsistência dos dados (HORSTMANN, 2016). Assim, a Java Virtual Machine (JVM), que faz a execução dos códigos junto aosistema operacional, consegue garantir que cada thread acesse os métodos de forma individual (FURGERI, 2015).

No Quadro 2.19 temos um exemplo do uso do mecanismo de sincronização. Nas linhas 2 e 3 foram utilizados métodos synchronized, com isso, quando diversas threads acessarem esses métodos, a própria JVM, junto ao sistema operacional, consegue garantir que o acesso seja individual.

```
    package U2S3;

    public class ContadorSync {
1.      private int quantidadeAlunosCurso;

2.      public synchronized void incrementa()
        {
          quantidadeAlunosCurso++;
        }

3.      public synchronized void decrementa()
        {
          quantidadeAlunosCurso--;
        }
}
```

Com os métodos synchronized, é possível produzir códigos paralelos confiáveis. Alguns cenários são propícios para o uso de threads, por exemplo, monitoramento dos recursos de rede, que acessam arquivos no disco de forma periódica, entre outras ações. Para a produção de uma thread que faça ações recorrentes, é necessário encapsular a ação dentro do método run(), como no Quadro 2.20.

No código do Quadro 2.20, na linha 1 é implementada a interface Runnable para que seja possível criar threads e utilizar o método run() da linha 9 (lembrando que ele é obrigatório, pois é por onde a thread começará a ser executada).

O item pausa na linha 5 (dentro do construtor da classe) define o intervalo que a ação da thread será executada que, nesse caso, foi definido como pausa = 1000;. Quando a thread é instanciada, th = new Thread(this), e iniciada, th.start(), o método
run() entra em ação. Nesse método, é necessário inserir o comando: Thread.sleep(pausa) logo no início e depois do processamento, para garantir que a thread não ocupe os núcleos de processamento, sem permitir que outros processos (como do sistema operacional) sejam executados. Usando essas pausas, é possível garantir que todas as threads façam seu processamento e, ao implementar os métodos com a diretiva synchronized, os resultados serão convergidos ao final. Porém, para a criação de threads ainda temos de pensar nas questões da orientação a objetos, nos aspectos da coesão e do acoplamento. No código do Quadro 2.20, a classe que controla a thread também faz o processamento e também é necessário fazer o controle manual do intervalo de processamento da thread utilizando o Thread.sleep(pausa);. Seguindo a ideia da orientação a objetos, vamos procurar uma forma mais adequada para fazer essas tarefas?

```
package U2S3;

1.  public class RepetidorThread implements Runnable{

2.      private int pausa;
3.      private boolean executa;
4.      private Thread th;

        public RepetidorThread()
        {
5.          pausa = 1000;
6.          executa = true;
7.          th = new Thread(this);
8.          th.start();
        }

9.      public void run() {
            try {
10.             Thread.sleep(pausa);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
11.         while(executa)
            {
                // faz o processamento
                try {
12.                 Thread.sleep(pausa);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
}
```

Anhanguera

Para evitar os problemas das questões de modelagem e ainda aumentar o acoplamento das classes que utilizam threads, é possível utilizar as classes Timer e TimerTak (HORSTMANN, 2016). A Timer é responsável por controlar a thread para que seja executada de forma periódica com um intervalo definido pelo usuário. A TimerTaks é abstrata e implementa a interface Runnable, ou seja, é necessário fazer uma especialização dela e implementar o método run(). Para que o sistema funcione, é preciso implementar as duas classes, pois na classe Timer o método que inicia o processamento espera uma instância da classe TimerTask. Com isso, a parte que cuida docontrole da thread (Timer) fica separada da que faz o processamento (TimerTask). Dessa maneira, a coesão das classes aumenta, criando formas de manutenção mais amplas.

Veja como utilizar as classes Timer e TimerTask no Quadro 2.21. A linha 1 import java.util.Timer; declara qual timer utilizar (não se deve confundir com o javax.swing.Timer), e a linha 2 declara o objeto timer do tipo Timer. Na linha 3 é declarado um objeto da classe RepetidorTimeTarefa. Essa classe é a especialização da TimerTask, que veremos logo a seguir. Na linha 4 foi criada uma variável chamada pausa, e esse valor será passado na classe Timer para controlar o tempo das threads. No construtor da classe public RepetidorTimer(), instanciamos a classe Timer e a tarefa do tipo RepetidorTimeTarefa e também definimos o valor da variável pausa = 1000 (em milissegundos). Por fim, configuramos a tarefa que o Timer fará. Em timer.schedule(tarefa, 0 , pausa);, o parâmetro tarefa (uma especialização da classe TimerTaks) define qual ação esse timer fará, o valor 0 define o atraso para iniciar a tarefa e o valor pausa define o intervalo em que a thread será executada.

Anhanguera

```
   package U2S3;
1. import java.util.Timer;


   public class RepetidorTimer{


2.      private Timer timer;
3.      private RepetidorTimeTarefa tarefa;
4.      private int pausa;
```

```
   public RepetidorTimer()
        {
5.        timer = new Timer();
6.        pausa = 1000;
7.        tarefa = new
8. RepetidorTimeTarefa("arquivoDados.txt");
          timer.schedule(tarefa, 0, pausa);
        }
        public static void main(String[] args) {
          RepetidorTimer rt = new RepetidorTimer();
        }
}
```

A classe do Quadro 2.22 representa o private RepetidorTimeTarefa tarefa; do Quadro 2.21 e tem o papel de especificar e executar a tarefa da thread. Vamos aproveitar e ver uma aplicação de um TimerTask? O Quadro 2.22 apresenta o uso da classe TimerTask. Veja que na linha 1 a classe RepetidorTimeTarefa faz uma especialização da TimerTask (extends). Com essa implementação, a classe RepetidorTimeTarefa é uma especialização da classe TimerTask, e a classe TimerTask implementa a interface Runnable. Dessa forma, a classe RepetidorTimeTarefa deve criar o método run(), como foi feito na linha 3. O método run() faz a busca de um arquivo na linha 5 e, na linha 6, ele verifica se o arquivo existe. Caso verdadeiro, nas linhas 7, 8 e 9 é feito o acesso ao arquivo e a leitura de seus dados, e na linha 10 é feita a impressão das linhas de texto do arquivo.

```
package U2S3;

import java.io.*;
import java.util.TimerTask;


1.  public class RepetidorTimeTarefa extends
    TimerTask {


2.    private String arquivo;
```

```
   public RepetidorTimeTarefa(String parquivo)
   {
      this.arquivo = parquivo;
   }
3.    public void run() {
      System.out.println("Buscando arquivo.");
      try {
5.       File f = new File(arquivo);
         System.out.println("Arquivo
6.  encontrado.");
         if(f.exists() == true)
         {
            String line = null;
7.          FileReader fileReader = new
    FileReader(f);
8.          BufferedReader bufferedReader =
9.                new BufferedReader(fileReader);
10.         while((line = bufferedReader.
    readLine())
                                        != null) {
               System.out.println(line);
            }
         }
      }catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Fonte: elaborado pelo autor.

O Quadro 2.23 apresenta o resultado do processamento feito de 1 em 1 segundo. Repare que a classe que faz o processamento RepetidorTimeTarefa não necessariamente precisa ser utilizada dentro de cenários de uma thread. Com isso, o acoplamento entre as classes fica baixo, portanto, aumenta as chances de reaproveitamento de código.

```
Buscando arquivo.
Buscando arquivo.
Buscando arquivo.
Buscando arquivo.
Arquivo encontrado.
Aluno1;10;10
Aluno2;9;9
Aluno3;9;9
```

Diversas linguagens de programação propõem maneiras diferentes de tratar o paralelismo. Algumas apresentam formas nas quais a complexidade é mais alta, como na linguagem C. A linguagem Java possui diversas formas de paralelismo para cenários mais simples ou mais complexos. Qual das opções abaixo apresenta recursos para o paralelismo em Java?

a) java.util.Timer e java.lang.Thread.
b) java.swing.JFrame e java.swing.JFrame.
c) java.lang.StringBuilder e java.lang.StrungBuffer.
d) java.util.Threads e java.lang.Parallel.
e) java.lang.Proc e java.swingx.Procstt.

A classe Timer do Java é muito útil pois encapsula o comportamento da classe Thread, tornando o processo da programação paralela em Java mais simples do que utilizar um sistema diretamente com as threads. Ainda, a utilização da classe Timer facilita nos processos de coesão e acoplamento de um sistema orientado a objetos, pois separa em duas classes as ações de controle da Thread e o processamento. As classes Timer e TimerTask possuem papéis diferentes. Quais são eles, respectivamente?

a) Processamento e controle de execução das threads.
b) Busca do horário de sistema e apresentação dos tempos.
c) Criação de pool de threads e processamento.
d) Remoção de dados e separação de dados.
e) Controle de execução das threads e processamento.