Módulo 2

Introdução à Programação II

Lição 9

Threads

Versão 1.0 - Mar/2007

Autor
Rebecca Ong

Equipe
Joyce Avestro
Florence Balagtas
Rommel Feria
Rebecca Ong
John Paul Petines
Sun Microsystems
Sun Philippines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Alexandre Mori
Alexis da Rocha Silva
Aline Sabbatini da Silva Alves
Allan Wojcik da Silva
André Luiz Moreira
Andro Márcio Correa Louredo
Antoniele de Assis Lima
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bruno da Silva Bonfim
Bruno dos Santos Miranda
Bruno Ferreira Rodrigues
Carlos Alberto Vitorino de Almeida
Carlos Alexandre de Sene
Carlos André Noronha de Sousa
Carlos Eduardo Veras Neves
Cleber Ferreira de Sousa
Cleyton Artur Soares Urani
Cristiano Borges Ferreira
Cristiano de Siqueira Pires
Derlon Vandri Aliendres
Fabiano Eduardo de Oliveira
Fábio Bombonato
Fernando Antonio Mota Trinta
Flávio Alves Gomes
Francisco das Chagas
Francisco Marcio da Silva
Gilson Moreno Costa
Givailson de Souza Neves
Gustavo Henrique Castellano
Hebert Julio Gonçalves de Paula
Heraldo Conceição Domingues
Hugo Leonardo Malheiros Ferreira
Ivan Nascimento Fonseca
Jacqueline Susann Barbosa
Jader de Carvalho Belarmino
João Aurélio Telles da Rocha
João Paulo Cirino Silva de Novais
João Vianney Barrozo Costa
José Augusto Martins Nieviadonski
José Leonardo Borges de Melo
José Ricardo Carneiro
Kleberth Bezerra G. dos Santos
Lafaiete de Sá Guimarães
Leandro Silva de Morais
Leonardo Leopoldo do Nascimento
Leonardo Pereira dos Santos
Leonardo Rangel de Melo Filardi
Lucas Mauricio Castro e Martins
Luciana Rocha de Oliveira
Luís Carlos André
Luís Octávio Jorge V. Lima
Luiz Fernandes de Oliveira Junior
Luiz Victor de Andrade Lima
Manoel Cotts de Queiroz
Marcello Sandi Pinheiro
Marcelo Ortolan Pazzetto
Marco Aurélio Martins Bessa
Marcos Vinicius de Toledo
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauricio Azevedo Gamarra
Mauricio da Silva Marinho
Mauro Cardoso Mortoni
Mauro Regis de Sousa Lima
Namor de Sá e Silva
Néres Chaves Rebouças
Nolyanne Peixoto Brasil Vieira
Paulo Afonso Corrêa
Paulo José Lemos Costa
Paulo Oliveira Sampaio Reis
Pedro Antonio Pereira Miranda
Pedro Henrique Pereira de Andrade
Renato Alves Félix
Renato Barbosa da Silva
Reyderson Magela dos Reis
Ricardo Ferreira Rodrigues
Ricardo Ulrich Bomfim
Robson de Oliveira Cunha
Rodrigo Pereira Machado
Rodrigo Rosa Miranda Corrêa
Rodrigo Vaez
Ronie Dotzlaw
Rosely Moreira de Jesus
Seire Pareja
Sergio Pomerancblum
Silvio Sznifer
Suzana da Costa Oliveira
Tásio Vasconcelos da Silveira
Thiago Magela Rodrigues Dias
Tiago Gimenez Ribeiro
Vanderlei Carvalho Rodrigues Pinto
Vanessa dos Santos Almeida
Vastí Mendes da Silva Rocha
Wagner Eliezer Roncoletta

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Regina Mariani -** Coordenadora responsável pela parte jurídica
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Vimos diversas classes que são executadas sequencialmente. Uma classe seqüencial refere-se a uma classe que possui um fluxo de execução. Ela tem um ponto de início de execução, uma seqüência de execução e o final da execução. Possui um único processo sendo executado.

Contudo, numa situação do mundo real, existe a necessidade de capturar processos concorrentes. Sendo que estes processos são executados simultaneamente. Neste ponto entram as threads.

Ao final desta lição, o estudante será capaz de:

- Definir o que são threads
- Enumerar os diferentes estados de uma thread
- Explicar o conceito de prioridade na thread
- Utilizar os métodos da classe *Thread*
- Criar suas próprias threads
- Utilizar sincronização para execução de threads concorrentes que sejam independentes umas das outras
- Permitir que as threads se comuniquem umas com as outras concorrentemente em sua execução
- Utilizar as utilidades da concorrência

2. Definição e básico de *threads*

2.1. Definição de Thread

A *thread* refere-se a um fluxo seqüencial simples de controle em uma classe. Para simplificar, pense em *threads* como mais de um processo simultâneo que serão executados paralelamente a partir de um determinado ponto. Considere os modernos sistemas operacionais que permitem rodar vários softwares ao mesmo tempo. Enquanto escrevemos um documento no computador utilizando o editor de texto, podemos escutar uma música, navegar na rede e o relógio do computador continua sendo atualizado. O sistema operacional instalado no seu computador permite o que definiremos como **multitarefa**. Um aplicativo pode executar diversos processos simultaneamente. Um exemplo deste tipo de aplicação é a **HotJava** do navegador *web*, que permite a navegação através de páginas enquanto realizamos downloads de algumas imagens, assistimos animações e ouvimos musicas.

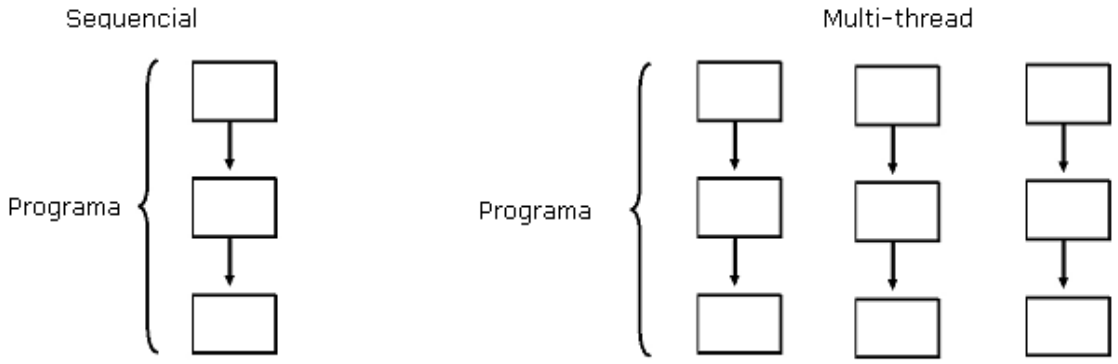

Figura 1: Threads

2.2. Estados da thread

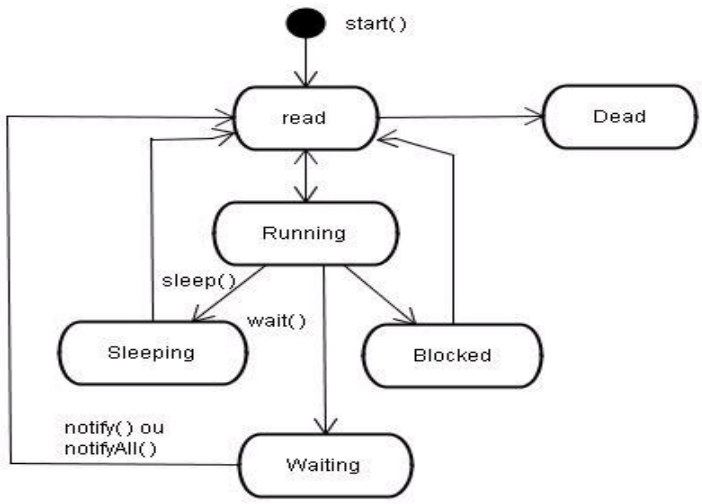

Figura 2: Estados da thread

Uma *thread* poderá assumir um dos seguintes estados:

- **Read**
  A thread pode ser executada, mas ainda não foi lhe dado à chance. Ou após ter sido bloqueada/suspensa, esta pronta para ser executada.

- **Running**
  A thread esta sendo executada e esta sob controle da CPU.
- **Sleeping**
  A thread é interrompida por determinado tempo.
- **Waiting**
  A thread é interrompida por indeterminado tempo.
- **Blocked**
  Uma thread bloqueada não é capaz de rodar, pois esta esperando por um recurso disponível ou que algum evento aconteça.
- **Dead**
  A thread terminou tudo o que foi ordenada a fazer.

2.3. Prioridades

É possível informar ao controle da CPU que uma *thread* possui uma prioridade maior sobre outra. A prioridade é um valor inteiro entre 1 e 10. Quanto maior a prioridade da *thread*, maior a chance de ser executado executada primeiro.

Assumiremos a existência de duas *threads* e supomos que ambas estejam prontas para rodar. A primeira *thread* é atribuído atribuída a prioridade 5 (normal) enquanto que a segunda possui a prioridade 10 (máxima). Digamos que a primeira *thread* esteja executando enquanto que que a segunda esteja pronta para ser executada. Então, a segunda *thread* poderia receber o controle da CPU e tornar-se executável por possuir a maior prioridade comparada com a *thread* corrente. Este cenário é um exemplo de troca de contexto.

A troca de contexto ocorre quando uma *thread* perde o controle da *CPU* para outra *thread*. Existem várias formas na qual a troca de contexto pode ocorrer. Um cenário é quando uma *thread* está sendo executada e voluntariamente cede o controle da *CPU* para que outra *thread* tenha oportunidade de ser executada. Neste caso, a *thread* de maior prioridade que esteja pronta para executar recebe o controle da *CPU*. Outra forma para que a troca de contexto ocorra é quando uma *thread* perde seu lugar para outra *thread* de maior prioridade como visto no exemplo anterior.

É possível que exista mais de uma thread disponível de maior prioridade e prontas para serem executadas. Para decidir qual das *threads* com a mesma prioridade receberá o controle da *CPU* isso dependerá do sistema operacional. No Windows 95/98/NT utilizará fracionamento de tempo e revezamento para resolver este caso. Cada *thread* de mesma prioridade é dado tempo limite para ser executada antes da *CPU* passar o controle para outra *thread* de igual prioridade. Por outro lado, o Solaris permite que *threads* sejam executadas antes de completarem sua tarefa, ou antes, de voluntariamente ceder o controle à CPU.

3. A classe Thread

3.1. Construtor

A classe *Thread* possui oito construtores. Vejamos rapidamente alguns destes.

Construtores da Thread
`Thread()`
Cria um novo objeto *Thread*.
`Thread(String name)`
Cria um novo objeto *Thread* com um nome específico.
`Thread(Runnable target)`
Cria um novo objeto *Thread baseado no objeto Runnable*. target refere-se ao objeto no qual o método executado será chamado.
`Thread(Runnable target, String name)`
Cria um novo objeto *Thread* com um nome específico e *baseado no objeto Runnable*.

Tabela 1: Construtores Thread

3.2. Constantes

A classe *Thread* provê constantes para valores de prioridade. Segue a tabela que nos mostra um resumo dos campos da classe *Thread*.

Constantes Thread
`public final static int MAX_PRIORITY`
O maior valor de uma prioridade, 10.
`public final static int MIN_PRIORITY`
O menor valor de uma prioridade, 1.
`public final static int NORM_PRIORITY`
O valor padrão de uma prioridade, 5.

Tabela 2: Constantes Thread

3.3. Métodos

Aqui estão alguns dos métodos encontrados na classe *Thread*.

Métodos Thread
`public static Thread currentThread()`
Retorna a referência para a thread corrente que esteja sendo executada.
`public final String getName()`
Retorna o nome da thread.
`public final void setName(String name)`
Renomeia a thread para o argumento especificado *name*. Poderá lançar *SecurityException.*
`public final int getPriority()`

Retorna a prioridade atribuída à thread.
`public final boolean isAlive()`
Indica se a thread esta executando ou não.
`public final void join([long millis, [int nanos]])`
Um método sobrecarregado. A thread corrente que esta sendo executada poderá esperar até que esta thread morra (se nenhum parâmetro for especificado), ou até que tempo especificado transcorra.
`public static void sleep(long millis)`
Suspende a thread por um tempo *millis*. Poderá lançar *InterruptedException*.
`public void run()`
A execução da thread inicia com este método.
`public void start()`
Ocasiona o início da execução da thread chamando o método *run*.

Tabela 3: Métodos Thread

3.4. Um Exemplo de Thread

Neste primeiro exemplo de uma *thread*, veremos como funciona um contador.

```
import javax.swing.*;
import java.awt.*;

class ThreadDemo extends JFrame {
   JLabel label;
   public ThreadDemo(String title) {
      super(title);
      label = new JLabel("Start count!");
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      this.add(label);
      this.setSize(300,300);
      this.setVisible(true);
   }
   private void startCount() {
      try {
         for (int i = 10; i > 0; i--) {
            Thread.sleep(500);
            label.setText(i + "");
         }
         label.setText("Count down complete.");
      } catch (InterruptedException ie) {
      }
      label.setText(Thread.currentThread().toString());
   }
   public static void main(String args[]) {
      new ThreadDemo("Count down GUI").startCount();
   }
}
```

A classe CountDownGui conta de 10 a 1 e depois mostra a informação sobre a *thread* que está sendo executado executada. Estas são algumas telas que foram retiradas enquanto a classe era executada:

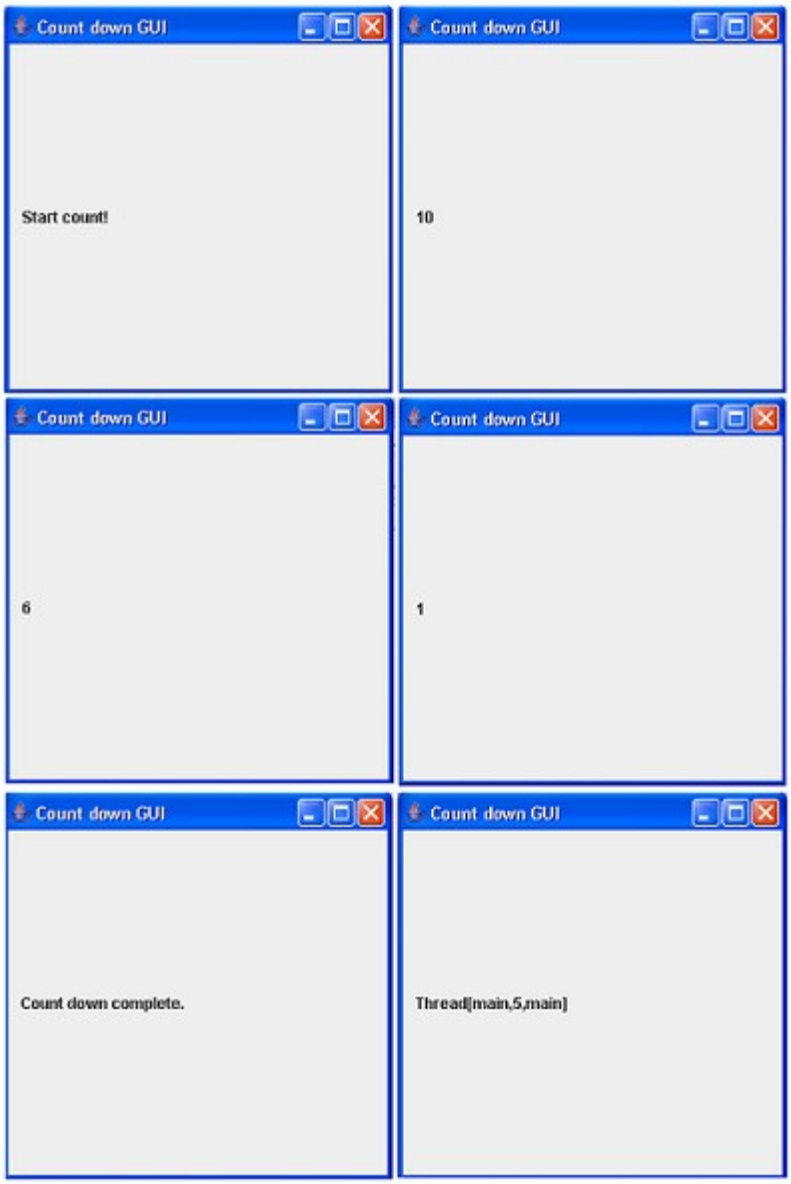

Figura 3: Telas da classe CountDownGui

4. Criando *Threads*

Threads podem tanto ser criadas de duas formas, a primeira é herdando a classe *Thread* (sua classe **é uma** *thread*) e a segunda é implementando a interface *Runnable* (sua classe **tem uma** *thread*).

4.1. Herança (estendendo) da classe Thread

Neste próximo exemplo, veremos uma classe que mostra 100 vezes o nome do objeto *thread*.

```
class ThreadDemo extends Thread {
   public ThreadDemo(String name) {
      super(name);
      start();
   }
   public void run() {
      for (int i = 0; i < 100; i++)
         System.out.print(getName());
   }
   public static void main(String args[]) {
      ThreadDemo pnt1 = new ThreadDemo("A");
      ThreadDemo pnt2 = new ThreadDemo("B");
      ThreadDemo pnt3 = new ThreadDemo("C");
      ThreadDemo pnt4 = new ThreadDemo("D");
   }
}
```

Observe que os objetos *pnt1*, *pnt2*, *pnt3*, e *pnt4* são utilizadas uma única vez. Para esta aplicação, não há qualquer necessidade de se utilizar atributos para se referir a cada *thread*. É possível substituir o corpo do método principal pelas seguintes instruções:

```
      new ThreadDemo("A");
      new ThreadDemo("B");
      new ThreadDemo("C");
      new ThreadDemo("D");
```

A classe produzirá diferentes saídas em cada execução. Aqui temos um exemplo de uma possível saída:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDAB
CDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCD
BCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDB
CDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCD
```

4.2. Implementando a interface Runnable

Outra forma do usuário para se criar uma *thread* é implementar a interface *Runnable*. O único método que a interface *Runnable* requer que seja implementado é o método "public void run()". Pense neste como o método que a *thread* executará.

O próximo exemplo é semelhante ao exemplo anterior mostrado:

```
class ThreadDemo implements Runnable {
   Thread thread;
```

```
    public ThreadDemo(String name) {
       thread = new Thread(this, name);
       thread.start();
    }
    public void run() {
       String name = thread.getName();
       for (int i = 0; i < 100; i++) {
          System.out.print(name);
       }
    }
    public static void main(String args[]) {
       new ThreadDemo("A");
       new ThreadDemo("B");
       new ThreadDemo("C");
       new ThreadDemo("D");
    }
}
```

4.3. É uma (extends Thread) x Tem uma (implements Runnable)

Estas são as duas únicas formas de se obter *threads*, escolher entre elas é uma questão de gosto ou de necessidade. Implementar por exemplo a interface *Runnable* pode ser mais trabalhoso já que ainda teremos que declarar um objeto *Thread* e comunicar os métodos *thread* sobre este objeto. Herdar a classe *Thread*, no entanto, significaria que a classe não poderia mais estender qualquer outra classe, pois Java proíbe heranças múltiplas. Uma troca entre implementação fácil e possibilidade de estender a classe é feita de acordo com o estilo selecionado. Pese o que é mais necessário.

4.4. Utilizando o método join

Agora que já vimos como criar *threads*, iremos ver como o método *join* funciona. O exemplo abaixo utiliza o método *join* sem qualquer argumento. Este método (quando chamado sem argumento) causa a espera da execução da *thread* corrente até que a *thread* que chama este método termine a sua execução.

```
class ThreadDemo implements Runnable {
   Thread thread;
   public ThreadDemo(String name) {
      thread = new Thread(this, name);
      thread.start();
   }
   public void run() {
      String name = thread.getName();
      for (int i = 0; i < 100; i++) {
         System.out.print(name);
      }
   }
   public static void main(String args[]) {
      ThreadDemo t1 = new ThreadDemo("A");
      ThreadDemo t2 = new ThreadDemo("B");
      ThreadDemo t3 = new ThreadDemo("C");
      ThreadDemo t4 = new ThreadDemo("D");
      System.out.println("Running threads...");
      try {
         t1.thread.join();
         t2.thread.join();
         t3.thread.join();
```

```
                        t4.thread.join();
                    } catch (InterruptedException ie) {
                    }
                    System.out.println("\nThreads killed."); //printed last!
                }
            }
```

Execute esta classe. O que notamos? Através da chamada do método *join*, temos a certeza que o último trecho foi executado até a última parte.

Aqui está um exemplo de saída da execução do código:

Running Threads…
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBCDBCDBCDBCDBCBCBCBCBCDB
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Threads killed.

Comente o bloco ***try-catch*** onde o método *join* foi executado. Existe alguma diferença na saída da classe?

5. Sincronização

Até agora, temos visto exemplos de *threads* que estão sendo executadas concorrentemente, mas são independentes umas da outras. Ou seja, *threads* que são executadas no seu próprio ritmo sem a preocupação pelo status ou pelas atividades de outras *threads* que são executadas concorrentemente. Desta forma, cada *thread* não requer qualquer fonte ou método externo e, como resultado, não precisa se comunicar com outras *threads*.

Em muitas situações, executar threads concorrentemente pode requerer fontes e métodos externos. Assim, existe a necessidade de se comunicar com outras *threads* executadas concorrentemente para saber seus status e atividades. Um exemplo neste cenário é o problema produtor-consumidor. O problema envolve dois objetos principais, um que funcionará como um produtor gerando valores ou dados e outro como consumidor para receber ou consumir estes valores.

5.1. Um exemplo não sincronizado

Vejamos a seguinte classe que mostra uma seqüência de *Strings* em uma ordem específica:

```
class TwoStrings {
   static void print(String str1, String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}
class PrintStringsThread implements Runnable {
   Thread thread;
   String str1, str2;
   PrintStringsThread(String str1, String str2) {
      this.str1 = str1;
      this.str2 = str2;
      thread = new Thread(this);
      thread.start();
   }
   public void run() {
      TwoStrings.print(str1, str2);
   }
}
class ThreadDemo {
   public static void main(String args[]) {
      new PrintStringsThread("Hello ", "there.");
      new PrintStringsThread("How are ", "you?");
      new PrintStringsThread("Thank you ", "very much!");
   }
}
```

É esperado que a classe mostre dois argumentos dos objetos *Runnable* consecutivamente. O problema, no entanto, é que a invocação do método *sleep* acarreta a execução de outras *threads* quando alguma outra *thread* ainda não terminou com a execução do método print da classe TwoStrings. Aqui está um exemplo de saída:

```
Hello How are Thank you there.
you?
```

```
very much!
```

Nesta execução, as três *threads* devem ter seus primeiros argumentos mostrados antes dos segundos argumentos. Isto resulta em uma saída oculta.

Na verdade, o este exemplo não coloca um problema muito sério, mas para outras aplicações, isto pode ocasionar algumas exceções ou problemas.

5.2. Travando um objeto

Para garantir que apenas uma *thread* tenha acesso a um método, Java permite travar um objeto incluindo seus métodos com o uso de monitores. O objeto entra com o monitor implícito quando a sincronização de métodos é invocada. Assim que um objeto estiver sendo monitorado, o monitor garante que nenhuma outra *thread* acesse o mesmo método. Como conseqüência, isso garante que apenas uma *thread* por vez irá executar o método do objeto.

Para sincronizar um método, utilizamos a palavra-chave *synchronized* na assinatura do método. Neste caso não podemos modificar o código fonte da definição do método, podemos sincronizar o objeto no qual o método pertence. A sintaxe para sincronizar um objeto é a seguinte:

```
synchronized (<object>) {
   //trecho de código a ser sincronizado
}
```

Com isso, os métodos do objeto podem ser invocados uma vez por *thread*.

5.3. Primeiro Exemplo Sincronizado

Aqui esta o código modificado onde o método *print* da classe *TwoString* está sincronizado.

```
class TwoStrings {
   synchronized static void print(String str1, String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}

class PrintStringsThread implements Runnable {
   Thread thread;
   String str1, str2;
   PrintStringsThread(String str1, String str2) {
      this.str1 = str1;
      this.str2 = str2;
      thread = new Thread(this);
      thread.start();
   }
   public void run() {
      TwoStrings.print(str1, str2);
   }
}

class ThreadDemo {
   public static void main(String args[]) {
      new PrintStringsThread("Hello ", "there.");
      new PrintStringsThread("How are ", "you?");
      new PrintStringsThread("Thank you ", "very much!");
```

```
   }
}
```

A classe agora produz os trechos corretos:

```
Hello there.
How are you?
Thank you very much!
```

5.4. Segundo Exemplo Sincronizado

Esta é uma outra versão da classe anterior. O método *print* da classe *TwoString* está sincronizado. Porém, ao invés de utilizar a palavra reservada *synchronized* no método, está sendo aplicado no objeto.

```
class TwoStrings {
   static void print(String str1, String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}

class PrintStringsThread implements Runnable {
   Thread thread;
   String str1, str2;
   TwoStrings ts;
   PrintStringsThread(String str1, String str2, TwoStrings ts)
   {
      this.str1 = str1;
      this.str2 = str2;
      this.ts = ts;
      thread = new Thread(this);
      thread.start();
   }
   public void run() {
      synchronized (ts) {
         ts.print(str1, str2);
      }
   }
}

class TestThread {
   public static void main(String args[]) {
      TwoStrings ts = new TwoStrings();
      new PrintStringsThread("Hello ", "there.", ts);
      new PrintStringsThread("How are ", "you?", ts);
      new PrintStringsThread("Thank you ", "very much!", ts);
   }
}
```

A classe também mostrará os trechos corretos.

6. Comunicação entre *threads*

Nesta seção, iremos aprender sobre os métodos básicos utilizados para permitir a comunicação entre *threads* concorrentementes enquanto são executadas.

Métodos para Comunicação entre threads
`public final void wait()`
Faz com que esta thread espere até que outra chame o método *notify* ou *notifyAll* neste objeto. Poderá lançar *InterruptedException*.
`public final void notify()`
Torna executável a thread que teve o método *wait* chamado neste objeto.
`public final void notifyAll()`
Torna executáveis as threads que tiveram o método *wait* chamado neste objeto.

Tabela 4: Método para Comunicação entre Threads

Para facilitar o entendimento destes métodos, vamos considerar o cenário garçom/cliente. No restaurante, o garçom *espera (waits)* que o cliente faça um pedido, ao invés de estar sempre perguntado se alguém deseja fazer um pedido. Quando o cliente entra, significa que o cliente está interessado na comida do restaurante. Desta forma, o cliente ao entrar no restaurante notifica (*notifies*) o garçom que irá necessitar dos seus serviços. Contudo, nem sempre o cliente está preparado para fazer um pedido. Seria chato se garçom ficasse continuamente perguntando ao cliente se o mesmo deseja fazer o pedido. Então, o garçom aguarda o cliente notificá-lo (*notifies)* que está pronto. Uma vez que o cliente entregou o pedido, seria um incomodo continuar importunando o garçom se o seu pedido já está entregue. Normalmente o cliente aguarda o garçom notificá-lo (*notifies)* servindo a comida.

Observe neste cenário que a parte que aguarda é executada assim que o cliente informa o garçom. O mesmo é verdade nas *threads*.

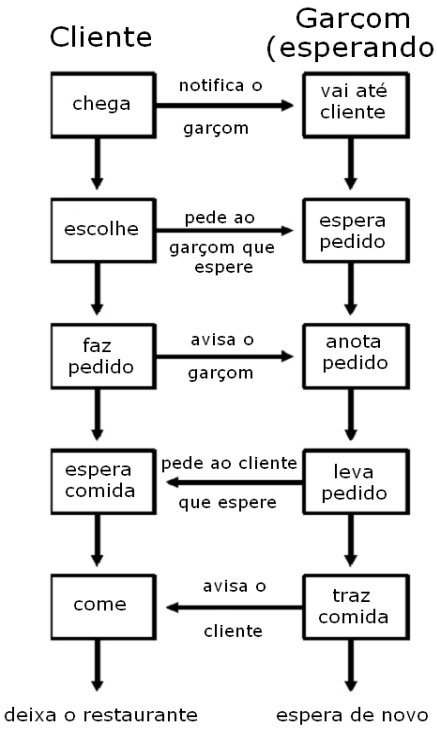

Figura 4: Cenário Cliente/Garçom

6.1. Exemplo Produtor-Consumidor

O seguinte exemplo é uma implementação do problema Produtor-Consumidor. A classe que provê métodos para gerar e consumir um valor inteiro são separadas as classes *threads* Produtor e Consumidor

```
class SharedData {
   int data;
   synchronized void set(int value) {
      System.out.println("Generate " + value);
      data = value;
   }
   synchronized int get() {
      System.out.println("Get " + data);
      return data;
   }
}

class Producer implements Runnable {
   SharedData sd;
   Producer(SharedData sd) {
      this.sd = sd;
      new Thread(this, "Producer").start();
   }
   public void run() {
      for (int i = 0; i < 10; i++) {
         sd.set((int)(Math.random()*100));
      }
   }
}

class Consumer implements Runnable {
   SharedData sd;
   Consumer(SharedData sd) {
      this.sd = sd;
      new Thread(this, "Consumer").start();
   }
   public void run() {
      for (int i = 0; i < 10 ; i++) {
         sd.get();
      }
   }
}
class ProducerConsumerDemo {
   public static void main(String args[]) throws Exception {
      SharedData sd = new SharedData();
      new Producer(sd);
      new Consumer(sd);
   }
}
```

Este é um exemplo da saída desta classe:

```
Generate 8
Generate 45
Generate 52
Generate 65
Get 65
Generate 23
```

```
Get 23
Generate 49
Get 49
Generate 35
Get 35
Generate 39
Get 39
Generate 85
Get 85
Get 85
Get 85
Generate 35
Get 35
Get 35
```

Isto não é o que gostaríamos que a classe produzisse. Para todos os valores produzidos pelo produtor, nós esperamos que o consumidor obtenha cada valor. Aqui está uma saída que seria desejável:

```
Generate 76
Get 76
Generate 25
Get 25
Generate 34
Get 34
Generate 84
Get 84
Generate 48
Get 48
Generate 29
Get 29
Generate 26
Get 26
Generate 86
Get 86
Generate 65
Get 65
Generate 38
Get 38
Generate 46
Get 46
```

Para consertar o problema com este código, utilizamos os métodos de comunicação entre *threads*. A seguinte implementação do problema Produtor-Consumidor utiliza os métodos para comunicação entre as *threads*.

```
class SharedData {
   int data;
   boolean valueSet = false;
   synchronized void set(int value) {
      if (valueSet) {   //has just produced a value
         try {
            wait();
         } catch (InterruptedException ie) {
         }
      }
      System.out.println("Generate " + value);
      data = value;
      valueSet = true;
      notify();
   }
```

```
   synchronized int get() {
      if (!valueSet) {  //producer hasn't set a value yet
         try {
            wait();
         } catch (InterruptedException ie) {
         }
      }
      System.out.println("Get " + data);
      valueSet = false;
      notify();
      return data;
   }
}

/* A parte remanescente do código não muda. */
class Producer implements Runnable {
   SharedData sd;
   Producer(SharedData sd) {
      this.sd = sd;
      new Thread(this, "Producer").start();
   }
   public void run() {
      for (int i = 0; i < 10; i++) {
         sd.set((int)(Math.random()*100));
      }
   }
}

class Consumer implements Runnable {
   SharedData sd;
   Consumer(SharedData sd) {
      this.sd = sd;
      new Thread(this, "Consumer").start();
   }
   public void run() {
      for (int i = 0; i < 10 ; i++) {
         sd.get();
      }
   }
}

class TestProducerConsumer {
   public static void main(String args[]) throws Exception {
      SharedData sd = new SharedData();
      new Producer(sd);
      new Consumer(sd);
   }
}
```

7. Utilidades concorrentes

Com a versão do J2SE 5.0, o novo controle de *threads* e das suas características concorrentes foram adicionados à Java. Estas novas características são encontradas no pacote *java.util.concurrent*. Nesta seção, duas destas características concorrentes serão vistas.

7.1. A Interface Executor

Uma das mais importantes características herdadas para desenvolver aplicações *multi-thread* é o *framework* Executor. A interface é incluída no pacote *java.util.concurrent*. Objetos deste tipo executam tarefas do tipo *Runnable.*

Sem o uso desta interface, executamos tarefas do tipo *Runnable* criando instâncias *Thread* e chamando o método *start* do objeto *Thread.* O seguinte código demonstra uma forma de fazer isso:

```
new Thread(<aRunnableObject>).start();
```

Com a disponibilidade desta nova interface, objetos *Runnable* submetidos são executados usando o método *execute* como a seguir:

```
<anExecutorObject>.execute(<aRunnableObject>);
```

O novo framework *Executor* é útil para aplicações *multithread*. Visto que a *thread* necessita da pilha de alocação dinâmica de memória, a criação de *threads* pode ser relativamente cara. Como resultado, a criação de várias *threads* pode causar um erro pela falta de memória. Uma forma de resolver este problema é com um *pool* de *threads*. No *pool* de *threads*, uma *thread* ao invés de morrer é enfileirada no *pool* após completar a sua tarefa designada. No entanto, implementar um esquema bem projetado de *pool* de *thread* não é simples. Outro problema é a dificuldade em cancelar e terminar as *threads*.

O *framework Executor* resolve estes problemas desacoplando à submissão de tarefas do mecanismo de como cada tarefa será executado executada, incluindo detalhes do uso de *threads*, agendamento e outros. Ao invés de criação e execução da *thread* através do métodos *start* para cada tarefa, ele é mais cauteloso, utilizar o seguinte fragmento de código:

```
Executor <executorName> = <anExecutorObject>;
<executorName>.execute(new <RunnableTask1>());
<executorName>.execute(new <RunnableTask2>());
...
```

Visto *Executor* ser uma interface, esta não pode ser instanciada. Para criar um objeto de *Executor*, deve-se criar uma classe implementando esta interface ou utilizar um método de fábrica disponível na classe *Executors.* Esta classe também está disponível no mesmo pacote da interface. A classe *Executors* provê métodos para o gerenciamento do *pool* de *threads*. Aqui esta um resumo destes métodos de fábrica:

Métodos de Fábrica da classe Executors
`public static ExecutorService newCachedThreadPool()`
Cria um pool de thread que cria novas thread threads quando necessário, porém reutilizará threads previamente construídas quando disponíveis. Existe um método sobrescrito, que aceita com como parâmetro um objeto ThreadFactory.
`public static ExecutorService newFixedThreadPool(int nThreads)`
Cria um pool de thread que reutiliza uma lista fixa de threads operando em segundo plano de uma fila ilimitada. Um método sobrescrito, o qual recebe um objeto ThreadFactory como parâmetro adicional.

`public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`
Cria um pool de thread que pode programar comando para serem executados após um período de tempo, ou periodicamente. Um método sobrescrito, o qual recebe um objeto ThreadFactory como parâmetro adicional.
`public static ExecutorService newSingleThreadExecutor()`
Cria um Executor que utiliza uma thread simples que trabalha operando em segundo plano de uma fila ilimitada. Um método sobrescrito, o qual recebe um objeto ThreadFactory como parâmetro adicional.
`public static ScheduledExecutorService newSingleThreadScheduledExecutor()`
Cria uma simples thread executora que pode programar comando a serem executados após um período de tempo, ou periodicamente. Um método sobrescrito, o qual recebe um objeto ThreadFactory como parâmetro adicional.

Tabela 5: Métodos de Fábrica da classe Executor

A tarefa *Runnable* é executada e finalizada sob o controle da interface *Executor*. Para parar *threads*, podemos invocar o método *shutdown* da interface como segue:

```
executor.shutdown();
```

7.2. A Interface Callable

Existem duas formas de se criar *threads*. Podemos tanto estender a classe *Thread* ou implementar a interface *Runnable*. Em qualquer técnica utilizada, customizamos as funcionalidades sobrescrevendo o método *run.* O método possui a seguinte assinatura:

```
public void run()
```

Os malefícios em se criar *threads* desde modo são:

1. O método *run* não permite retornar um resultado, pois seu tipo de retorno é *void.*
2. Requer que seja capturado as exceções verificáveis, visto ser um método a ser implementado e não possuir a cláusula *throws.*

A interface *Callable* é basicamente a mesma da interface *Runnable* sem estes problemas. Devemos utilizar outro mecanismo para obter o resultado de uma tarefa *Runnable*. Uma técnica seria utilizar uma variável de instância para armazenar o resultado. A classe a seguir mostra como isso pode ser feito.

```
public MyRunnable implements Runnable {
  private int result = 0;
  public void run() {
    ...
    result = someValue;
  }
  public int getResult() {
    return result;
  }
}
```

Vejamos como obter o resultado com a utilização da interface *Callable*:

```
import java.util.concurrent.*;

public class MyCallable implements Callable {
  public Integer call() throws java.io.IOException {
    ...
```

```
    return someValue;
  }
}
```

O método *call* possui a seguinte assinatura:

```
            <V> call() throws Exception
```

Onde <V> é um tipo genérico, que significa que o tipo de retorno do método *call* poderá ser uma referencia a qualquer tipo. Futuramente iremos aprender mais sobre tipos genéricos (*generics*).

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.