Módulo 8

Sistema Operacional

Lição 5

Java Thread

Versão 1.0 - Mar/2008

Autor
-

Equipe
Rommel Feria
John Paul Petines

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

Aécio Júnior
Alberto Ivo da Costa Vieira
Alexandre Mori
Alexis da Rocha Silva
Allan Wojcik da Silva
Antonio José Rodrigues Alves Ramos
Angelo de Oliveira
Bruno da Silva Bonfim
Carlos Fernandes Gonçalves
Denis Mitsuo Nakasaki
Felipe Gaúcho
Jacqueline Susann Barbosa
João Vianney Barrozo Costa
Luiz Fernandes de Oliveira Junior
Marco Aurélio Martins Bessa
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Paulo Oliveira Sampaio Reis
Ronie Dotzlaw
Seire Pareja
Thiago Magela Rodrigues Dias
Vinícius Gadis Ribeiro

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
- **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™

Original desta por – McDougall e Mauro – Solaris Internals. Sun Microsystems. 2007.

1. Objetivos

Nesta lição discutiremos sobre *Threads* na linguagem Java. Veremos os comandos de sincronização e soluções para os problemas de sincronização discutidos na lição anterior.

Ao final desta lição, o estudante será capaz de:

- Criar *Threads* em Java
- Fazer o uso da palavras-chave *synchronized*
- Utilizar os métodos *wait* e *notify* de *java.lang.Object*
- Manipular objetos de concorrência de alto nível

2. Criando uma Thread em Java

Thread, em linguagem Java, é a capacidade de diferentes partes de seu programa poder ser executada de modo simultâneo. Por exemplo, pode-se criar uma aplicação que aceite entradas de diferentes usuários ao mesmo tempo, cada um deles manipulando uma *thread*. A maior parte de aplicações de rede envolvem *threads*. Poderíamos necessitar criar uma *thread* que espere por uma determinada entrada enquanto o programa gera um relatório de saída.

Há duas maneiras para se criar *threads*, por herança da classe *thread* ou pela implementação de uma interface chamada *Runnable*. Ao ser iniciada, a *thread* executa as instruções contidas no método *run*().

2.1. Estendendo a classe Thread (is a)

Criaremos uma *thread* que mostrará 500 vezes um determinado número passado como argumento pelo construtor da classe.

```
class MyThread extends Thread {
  private int i;
  MyThread(int i) {
    this.i = i;
  }
  public void run() {
    for (int ctr=0; ctr < 500; ctr++) {
      System.out.print(i);
    }
  }
}
```

Primeiramente veremos a diferença entre uma execução paralela e não-paralela. Vejamos a seguinte classe:

```
class MyThreadDemo {
  public static void main(String args[]) {
    MyThread t1 = new MyThread(1);
    MyThread t2 = new MyThread(2);
    MyThread t3 = new MyThread(3);
    t1.run();
    t2.run();
    t3.run();
    System.out.print("Main ends");
  }
}
```

Conforme visto, após a execução da classe *MyThreadDemo*, foi chamado o método *run* do objeto **t1** que mostra 500 números **1**. Em seguida, a chamada ao método *run* do objeto **t2** que mostra 500 números **2**. Por fim, a chamada ao método *run* do objeto **t3** que mostra 500 números **3**. O texto "Main ends" aparece na parte final, como última mensagem da execução desta classe. Não ocorre nenhuma execução simultânea, isto é o que denominamos de execução não-paralela, conforme pode ser visto na figura a seguir.

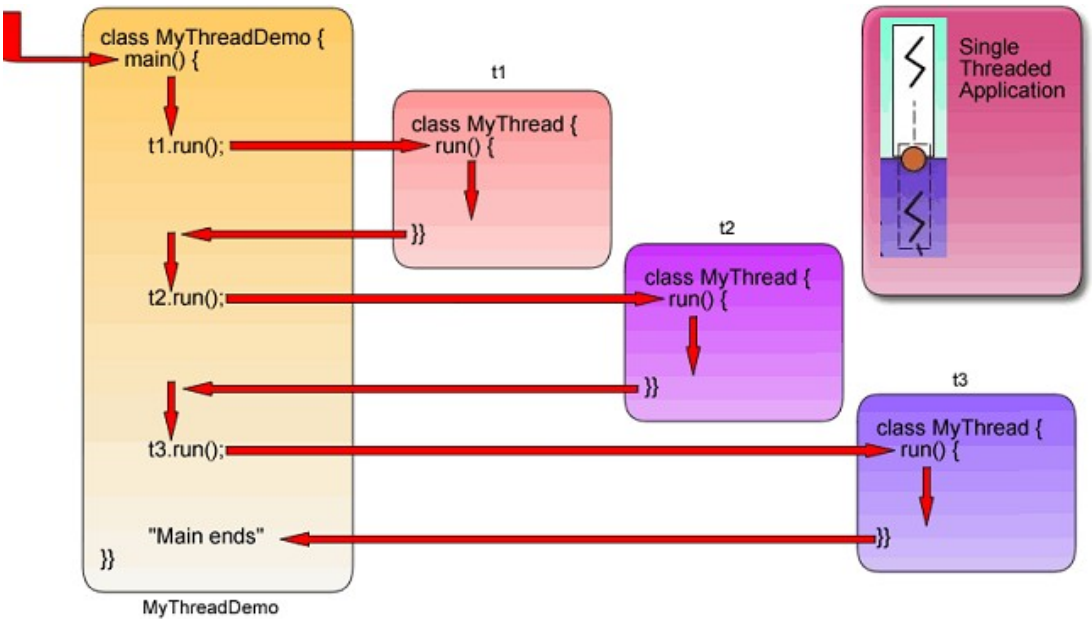

Figura 1: Execução não-paralela

Para executar através das *threads*, chamamos o método *start*() em vez de chamar diretamente o método *run*().

```
class MyThreadDemo {
  public static void main(String args[]) {
    MyThread t1 = new MyThread(1);
    MyThread t2 = new MyThread(2);
    MyThread t3 = new MyThread(3);
    t1.start();
    t2.start();
    t3.start();
    System.out.print("Main ends");
  }
}
```

Como podemos ver ao executar esta classe, os objetos *threads* **t1**, **t2** e **t3** agora estão rodando de modo simultâneo e os números **1**, **2** e **3** são mostrados intercaladamente. Interessante observar o aparecimento do texto "Main ends" no meio da seqüência de saída. Isso indica que o método *main* continua sendo executado enquanto as *threads* estão rodando.

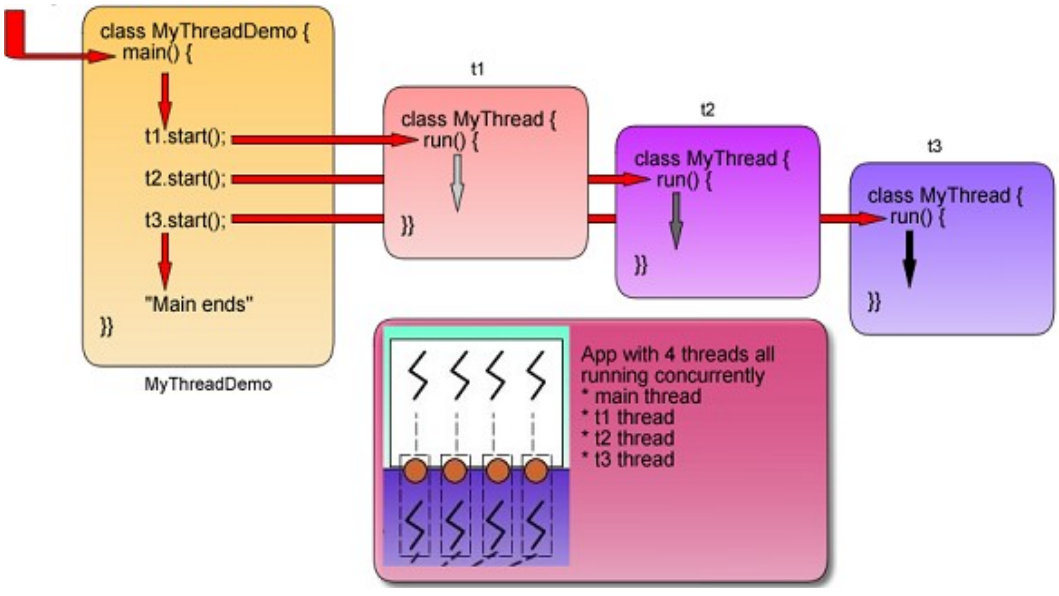

Figura 2: Execução das Threads

2.2. Implementando a interface *Runnable* (has a)

Outra maneira para se obter os benefícios de uma *thread* é implementar a interface *runnable*. Isto pode ser útil se sua classe necessita de herança. Lembre-se que a linguagem Java não permite herança múltipla. Modificamos nossa classe *MyThread* para implementar a interface *runnable*. Esta interface possui um único método abstrato que, obrigatoriamente, deve ser implementar, o método *public void run*().

```
class MyThread implements Runnable {
... <thread body is mostly the same>
}
```

A diferença principal está na construção do objeto da classe *Thread* que possuem um objeto da classe *MyThread*.

```
Thread t1 = new Thread(new MyThread(1));
```

Um objeto de *MyThread* agora é passado como um argumento para o construtor de um objeto *Thread*.

2.3. Pausar threads

Threads podem ser pausadas pelo método *sleep*(). Por exemplo, para interromper a execução de *MyThread* por meio segundo antes de imprimir o próximo número, adiciona-se as seguintes linhas de código:

```
for (int ctr=0; ctr < 500; ctr++) {
  System.out.print(i);
  try {
    Thread.sleep(500); // 500 milissegundos
  } catch (InterruptedException e) { }
}
```

O método *sleep(long time)* é estático dentro da classe *Thead* e pode ser invocado por qualquer *thread*, inclusive a do método principal. Por exemplo, se quisermos pausar por um segundo o processo antes de iniciar o objeto **t2** no nosso método principal, podemos ter:

```
public static void main(String args[]) {
  ...
  t1.start();
  try {
    Thread.sleep(1000);
  } catch (InterruptedException e) { }
  t2.start();
  ...
}
```

2.4. Concordância de Threads

É possível interromper uma *thread* até que outra termine sua execução, chamamos, para isso, o método *join*(). Por exemplo, para fazer a *thread* principal parar de rodar até que o objeto **t1** termine, podemos escrever:

```
public static void main(String args[]) {
  ...
  t1.start();
  try {
    t1.join();
  } catch (InterruptedException e) { }
  t2.start();
  ...
}
```

Isto determina que a *thread* **t1** termine de executar antes de iniciar a *thread* **t2**.

3. Palavra-chave Synchronized

Implementaremos uma solução para um problema de **Sessão Crítica** usando Java. Lembre-se que somente um processo pode entrar na sessão crítica do sistema, todos os outros processos devem aguardar. Nenhum chaveamento de contexto é permitido na *s*essão crítica.

Vejamos o seguinte problema: em vez de mostrar uma contínua sucessão de números, *MyThread* chama o método *print10*() na classe *MyPrinter*. O método *print10*() imprime números contínuos em uma única linha antes de iniciar uma nova linha.

Nossa meta é ter estes 10 números contínuos impressos sem que ocorra nenhum chaveamento de contexto. Em outras palavras, nossa saída deve ser:

```
...
1111111111
1111111111
2222222222
3333333333
1111111111
...
```

3.1. Definir a solução

A seguir, definimos a classe *MyPrinter* que possui o método *print10*():

```
class MyPrinter {
  public void print10(int value) {
    for (int i = 0; i < 10; i++) {
      System.out.print(value);
    }
    System.out.println(""); // uma nova linha após 10 números
  }
}
```

Em vez de imprimir os números diretamente, utilizamos o método *print10*() na classe *MyThread*, como mostrado a seguir:

```
class MyThread extends Thread {
  int i;
  MyPrinter p;
  MyThread(int i) {
    this.i = i;
    p = new MyPrinter();
  }
  public void run() {
    for (int ctr=0; ctr < 500; ctr++) {
      p.print10(i);
    }
  }
}
```

Visualizamos a saída de uma única *thread*.

```
class MyThreadDemo {
  public static void main(String args[]) {
    MyThread t1 = new MyThread(1);
    // MyThread t2 = new MyThread(2);
    // MyThread t3 = new MyThread(3);
    t1.start();
    // t2.start();
    // t3.start();
    System.out.print("Main ends");
  }
}
```

Ao ser executada a classe *MyThread*, teremos:

```
> java MyThreadDemo
1111111111
1111111111
1111111111
1111111111
1111111111
...
```

Contudo, ao rodar as outras *threads*, podemos ter uma saída semelhante a esta:

```
> java MyThreadDemo
1111111111
111112222222
1111
22233333332
...
```

Não alçamos nosso objetivo de imprimir os 10 números de forma consecutiva quando todas as *threads* executam o método *print10* ao mesmo tempo. O resultado final é que teremos mais de um número aparecendo em uma única linha.

Não deve haver um interruptor do contexto ao mostrarmos os 10 números consecutivos. Assim, uma solução para este problema é também uma solução para o problema de *s*essão crítica.

3.2. Monitores em Java

Java usa uma construção de monitor para resolver o problema de *s*essão crítica. Somente uma única *thread* pode funcionar dentro de um monitor. Para alternar um objeto em um monitor, colocamos a palavra-chave *synchronized* sobre as assinaturas dos métodos. Somente uma única *thread* pode executar em um método sincronizado dentro de um objeto.

```
class MyPrinter {
  public synchronized void print10(int value) {
    for (int i = 0; i < 10; i++) {
      System.out.print(value);
    }
    System.out.println(""); // nova linha depois de 10 números
  }
}
```

Entretanto, se transformarmos nosso método *print10*() em um método sincronizado, ainda assim este não trabalhará como esperado. Para encontrar um modo de fazê-lo funcionar corretamente, necessitamos descobrir como a palavra-chave *synchronized* trabalha.

Cada objeto em Java possui um bloqueio. Quando uma *thread* é executada em um método sincronizado, o objeto é bloqueado. Ao obter um bloqueio a *thread* começa a funcionar no método sincronizado, caso contrário, espera até que a *thread* que possui o bloqueio seja liberada.

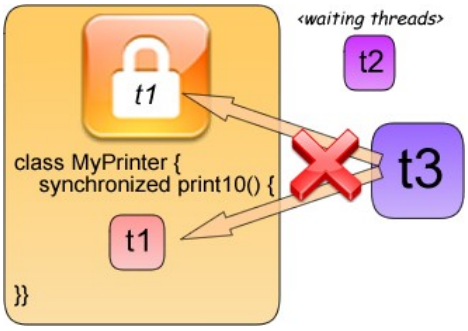

Figura 3: Método sincronizado

Nosso exemplo não trabalha corretamente, pois cada *thread* possui uma cópia própria do objeto *MyPrinter*.

```
class MyThread extends Thread {
  int i;
  MyPrinter p;
  MyThread(int i) {
    this.i = i;
    p = new MyPrinter(); // cada MyThread cria sua MyPrinter!
  }
  public void run() {
    for (int ctr=0; ctr < 500; ctr++) {
      p.print10(i);
    }
  }
}
```

Como podemos ver na figura a seguir, cada *thread* possui seu próprio bloqueio, deste modo, todas funcionam através de um método sincronizado:

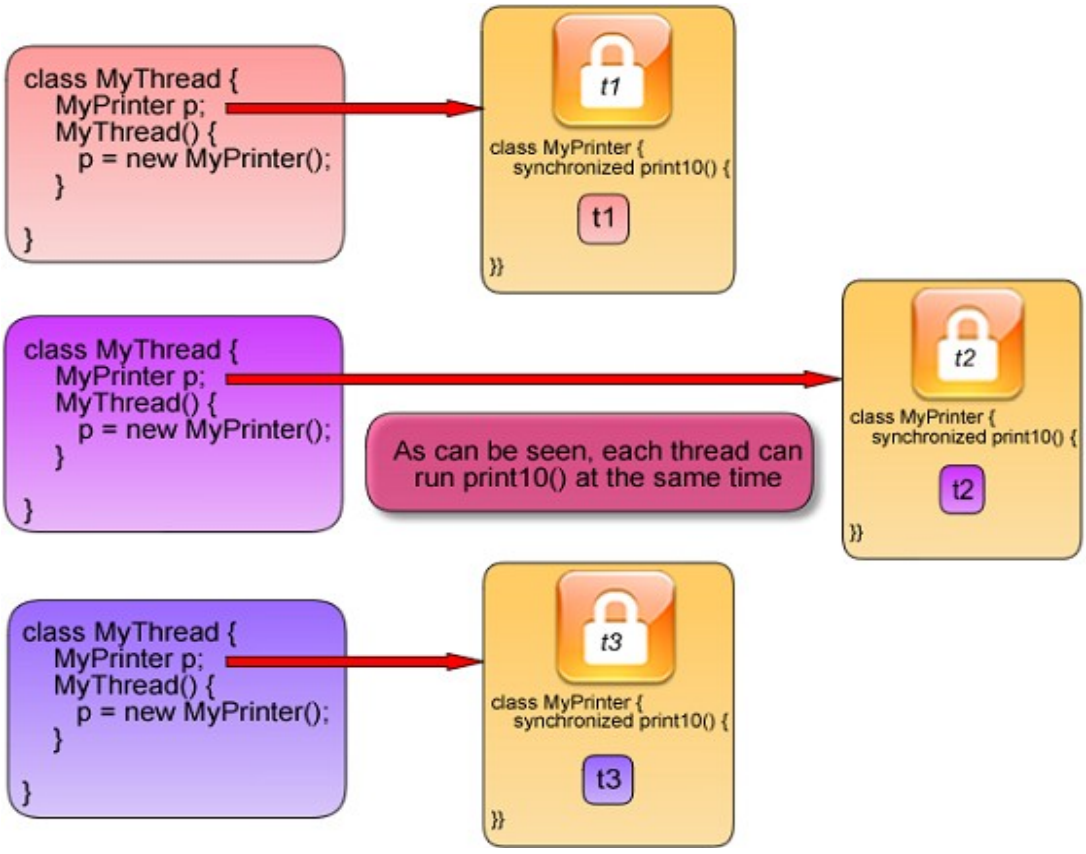

Figura 4: Diagrama dos objetos

A solução é um único objeto *MyPrinter* para compartilhamento por todas as *threads*.

```
class MyThread extends Thread {
      int i;
      MyPrinter p;
      MyThread(int i, MyPrinter p) {
            this.i = i; this.p = p
      }
      public synchronized void run() {
            for (int ctr=0; ctr < 500; ctr++) {
                  p.print10(i);
            }
      }
}
```

```
class MyThreadDemo {
      public static void main(String args[]) {
            MyPrinter p = new MyPrinter();
            MyThread2 t1 = new MyThread2(1,p);
            MyThread2 t2 = new MyThread2(2,p);
            MyThread2 t3 = new MyThread2(3,p);
            t1.start();
            t2.start();
            t3.start();
}}
```

Figura 5: Solução com um único objeto

Uma maneira de visualizar todos os objetos é através de portas. Uma *thread* tenta ver se uma

porta está aberta. Uma vez que a *thread* atravessa esta porta, esta é travada. Nenhuma outra *thread* pode entrar nesta porta porque a primeira a travou por dentro. As outras *thread* podem entrar se a *thread* que está dentro destravar a porta e sair. Um alinhamento ocorrerá somente se todos os objetos tiverem uma única porta na sessão crítica.

Reinterando, somente uma única *thread* pode rodar no método sincronizado de um objeto. Se um objeto tiver um método sincronizado e um método não-sincronizado, somente uma *thread* pode rodar no método sincronizado e múltiplas *threads* podem rodar no método não-sincronizado. *Threads* a serem sincronizadas devem compartilhar o mesmo objeto monitor. Ao executar agora a classe *MyThreadDemo*, o resultado é mostrado corretamente através dos métodos sincronizados.

3.3. Blocos sincronizados

Além dos métodos sincronizados, Java permite **blocos sincronizados**. Podemos especificar um bloqueio intrínseco. Os blocos sincronizados permitem a flexibilidade de que o bloqueio intrínseco venha de outro objeto ao invés do objeto atual. Também podemos ter partes de um método que possui um bloqueio diferente de outro.

Considere a seguinte classe *MyPrinter*:

```
class MyPrinter {
  Object lock1 = new Object();
  Object lock2 = new Object();
  public void print10(int value) {
    synchronized(lock1) {
      for (int i = 0; i < 10; i++) {
        System.out.print(value);
      }
      System.out.println(""); // nova linha após 10 números
    }
  }
  public int squareMe(int i) {
    synchronized (lock2) {
      return i * i;
    }
  }
}
```

Somente uma *thread* pode rodar dentro do bloco sincronizado dentro dos métodos *print10*() e *squareMe*(). Entretanto, duas *threads* diferentes podem rodar em ambos os blocos sincronizados ao mesmo tempo pois usamos bloqueios diferentes para cada um deles (porque não interferem realmente um com o outro). Os blocos sincronizados permitem também que outra parte de um método sejam executadas em paralelo com outras, como mostrado na figura a seguir:

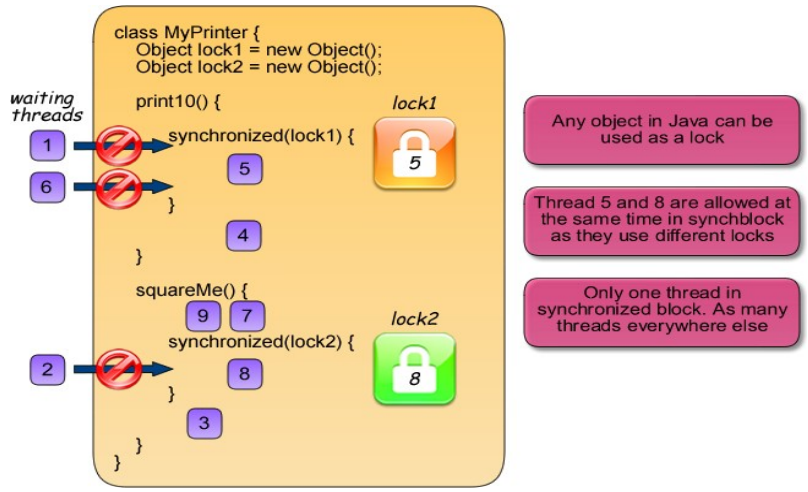

Figura 6: Diagrama de Sincronização

4. Métodos wait e notify

Neste ponto, já é possível utilizar algumas implementações para problemas relacionados a sincronização. Discutiremos o problema do **produtor-consumidor**. O **produtor** irá armazenar em um *array* compartilhado um valor inteiro gerado de forma aleatória que será obtido pelo **consumidor**. Para propósitos de discussão, será permitido que este *array* armazene um número muito grande de números inteiros. O **produtor** irá produzir, desta forma, um total de 100 números inteiros.

4.1. Blocos sincronizados

Primeiramente, criamos a classe para que possa ser realizado o armazenamento do *array*. Não será aceito que os métodos *insertValue*() e *getValue*() sejam executados simultaneamente. Assim, estes métodos serão sincronizados (terão o modificador *synchronized*). Tanto o **produtor** quanto o **consumidor** estarão tratando do mesmo objeto.

```
class SharedVars {
  int array[] = new int[100];
  int top;

  // Método responsável pela atribuição do valor
  public synchronized void insertValue(int value) {
    if (top < array.length) {
      array[top] =  value;
      top++;
    }
  }
  // Método responsável por recuperar o valor
  public synchronized int getValue() {
    if (top > 0) {
      top--;
      return array[top];
    } else {
      return -1;
    }
  }
}
```

A seguir, será definida a classe que representa o **produtor**.

```
class Producer extends Thread {
  SharedVars sv;
  Producer(SharedVars sv) {
    // Referência de Produtor para os objetos SharedVars
    this.sv = sv;
  }
  public void run() {
    for (int i = 0; i < 100; i++) {
      // Atribui um número aleatório entre 0 a 200
      int value = (int)(Math.random() * 200);
      System.out.println("Producer inserts " + value);
      sv.insertValue(value);
      try {
        // Espera um determinado tempo antes de realizar uma nova inserção
        Thread.sleep((int)(Math.random() * 10000));
      } catch (InterruptedException e) { }
    }
  }
}
```

E a classe que representa o **consumidor**:

```
class Consumer extends Thread {
  SharedVars sv;
  Consumer(SharedVars sv) {
```

```
    // Referência de Consumidor para os objetos SharedVars
    this.sv = sv;
  }
  public void run() {
    for (int i = 0; i < 100; i++) {
      int value = sv.getValue();
      System.out.println("Consumer got:" + value);
      try {
        // Espera um tempo antes antes de obter o valor novamente
        Thread.sleep((int)(Math.random() * 10000));
      }catch (InterruptedException e) { }
    }
  }
}
```

Para finalizar, temos a seguinte classe principal:

```
class ProducerConsumerDemo {
  public static void main(String args[]) {
    SharedVars sv = new SharedVars();
    Producer p = new Producer(sv);
    Consumer c = new Consumer(sv);
    p.start();
    c.start();
  }
}
```

A execução desta classe pode gerar a seguinte saída:

```
Producer inserts value: 15
Consumer got: 15
Consumer got: -1
Producer inserts value: 50
Producer inserts value: 75
Consumer got: 75
...
```

É importante notar que, caso seja realizada uma tentativa de recuperar um valor em uma posição no *array* que não possua nenhum valor, será obtido o valor **-1**. É possível notar isto se o **produtor** gerar um novo valor a cada 10 segundos e caso o **consumidor** recuperar este valor a cada 5 segundos. Em algum ponto o consumidor irá tentar realizar o acesso a uma posição vazia do *array*.

Então, é melhor que o **consumidor** espere até que o **produtor** gere algum valor no lugar de obter o valor **-1**.

4.2. método wait()

O método *wait*() é definido na classe *java.lang.Object* e é herdado por todos os objetos. A chamada a este método fará com que a *thread* suspenda suas atividades. Entretanto, se uma *thread* realizada uma chamada a este método, esta possuir um bloqueio para o objeto que está realizando esta chamada. Se for realizada uma chamada na forma this.wait() este deve estar em um bloco sincronizado, da seguinte forma:

```
synchronized(this);
```

De modo semelhante, outros métodos que realizam uma interrupção da execução de uma *thread*, tais como, *join*() e *sleep*(), o método *wait*() deve estar protegido por um bloco *try-catch* que controle a exceção tipo *InterruptedException*.

O código a seguir mostra a modificação para o método *getValue*() na classe *SharedVars*:

```
// chamado pela thread consumidor
public int getValue() {
  synchronized(this) {
    if (top <= 0) {
```

```
            try {
              this.wait();
            } catch (InterruptedException e) { }
          }
          top--;
          return array[top];
    }}
```

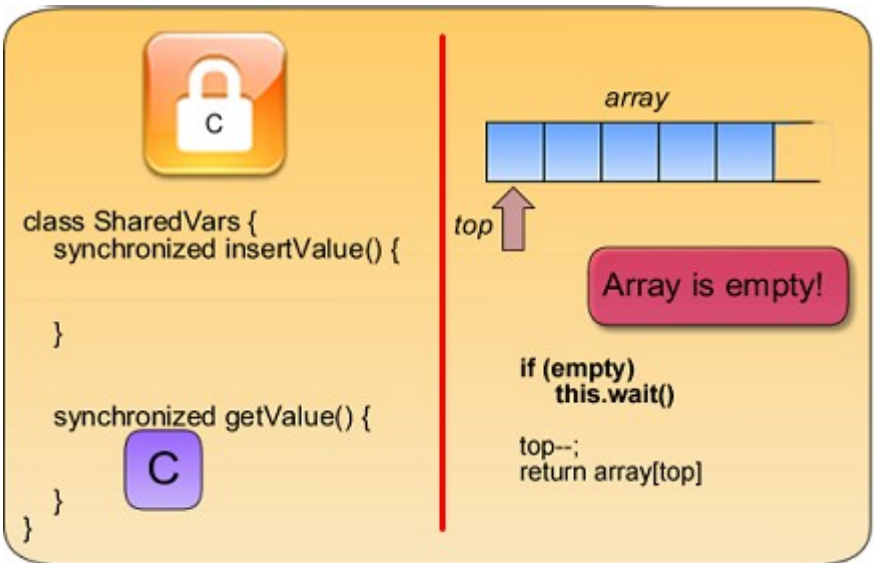

Figura 7: Diagrama de Sincronização

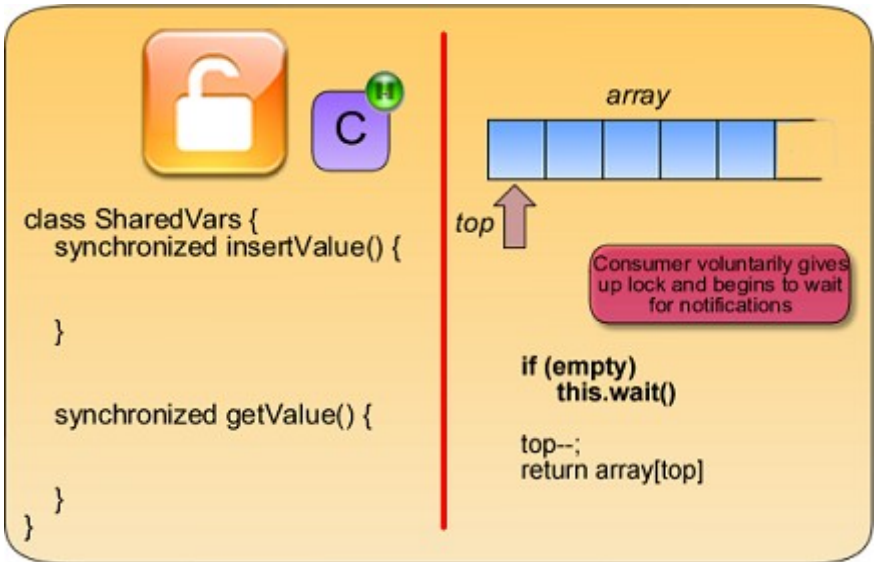

Figura 8: Diagrama de Sincronização

4.3. método notify()

Todas as *threads* que invocam o método *wait*() em um objeto são posicionadas em um agrupamento de *threads*. O retorno de uma *thread* interrompida é feito quando outra *thread* que está executando chama o método *notify*() do mesmo objeto. Quando isso é feito a primeira *thread* do *pool* retoma suas atividades.

Por exemplo, após o **produtor** realizar uma atribuição no *array*, pode notificar o **consumidor** pela chamada ao método *notify*() no objeto *SharedVars*. Lembrando de que tanto **produtor** quanto **consumidor** possuem uma mesma referência para o objeto *SharedVars*. Ou seja, o **produtor** invoca o método *notify*() a partir do método *insertValue*() e informa ao **consumidor** que se encontra em espera através do objeto *SharedVars*.

A seguir, teremos o novo método *insertValue*():

```
// Chamado pela Produtor
public void insertValue(int value) {
  synchronized(this) {
    if (top < array.length) {
      array[top] =  value;
      top++;
      if (top == 1) { // Se o array estiver vazio
        this.notify();
      }
    }
  }
}
```

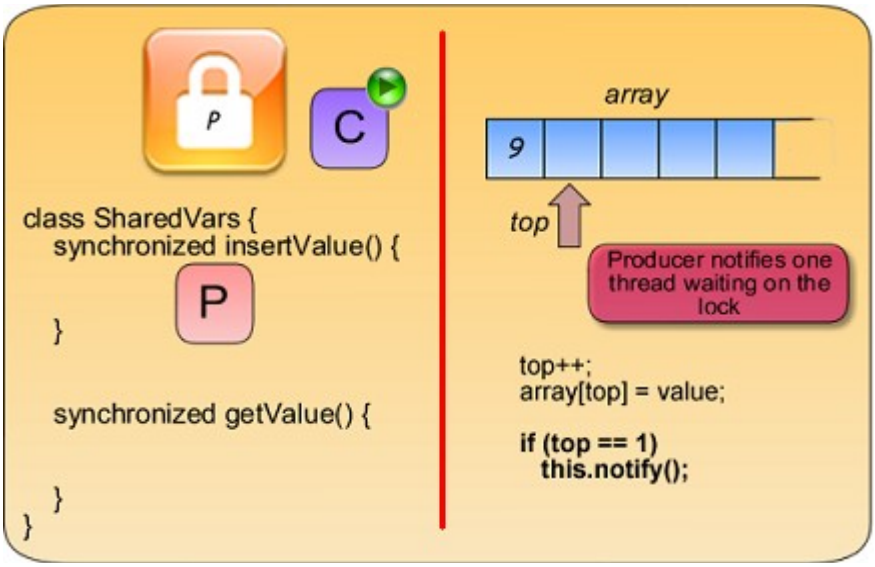

Figura 9: Diagrama de Sincronização

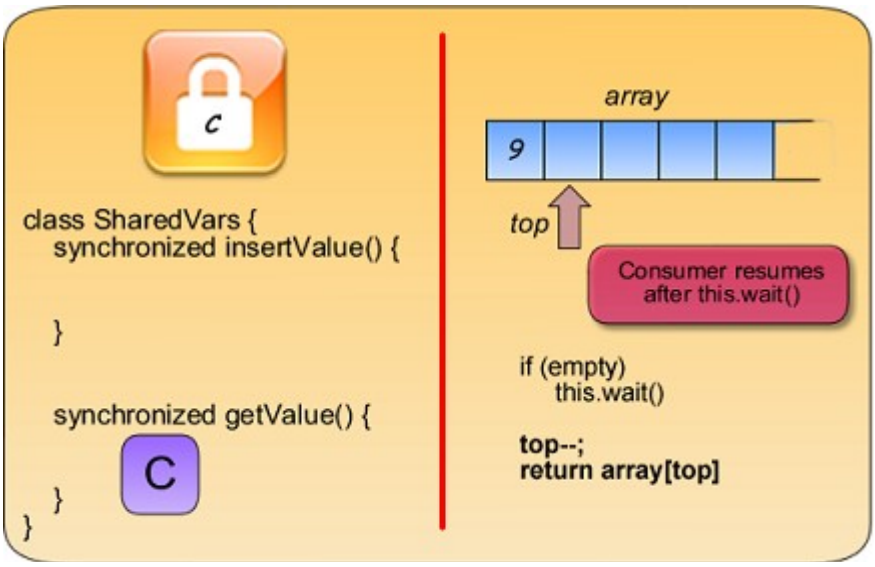

Figura 10: Diagrama de Sincronização

Ao executar novamente o exemplo, podemos observar que, mesmo se **produtor** estiver lento, o **consumidor** será obrigado a esperar até que seja gerado um valor antes de obtê-lo.

Devemos considerar também que podemos ter um *array* lotado, desta forma, será a vez do **produtor** esperar até que **consumidor** obtenha alguns valores antes de gerar novos valores. Realize esta implementação como uma forma de exercício.

5. Alto Nível de Concorrência dos Objetos

Os métodos discutidos até o momento, foram inseridos desde as primeiras versões da linguagem Java. Entretanto, com a chegada da Java 5.0 novas classes foram adicionadas ao pacote *java.util.concurrent* para auxiliar na criação e manipulação de *threads*.

5.1. Bloquear Objetos

Qualquer objeto em linguagem Java pode ser usado para efetuar um bloqueio durante uma sincronização. Porém, o pacote *java.util.concurrent.locks* define classes que possuem adicionais características para proceder este bloqueio. A interface básica deste pacote é a *Lock*.

Um objeto que implementa *Lock* só pode ser utilizado por uma única *thread* de cada vez. Para efetuar um bloqueio, a *thread* deve chamar o método *lock*(). Para liberar este bloqueio, deve chamar o método *unlock*(). Em vez de fechar implicitamente e destravar um bloco sincronizado ou método, um bloqueio explícito provê a flexibilidade de chamar manualmente os métodos *lock*() e *unlock*().

Um bloqueio poderá também chamar um método *tryLock*(). Este método retorna um valor lógico. Verdadeiro se conseguiria realizar um bloqueio, ou falso se não. O programa não irá bloquear enquanto *tryLock*() não obtiver sucesso, diferente do método *lock*() e do comportamento de uma *thread* que tenta entrar em um bloco ou método sincronizado. Além disso, um polimórfico do método *tryLock*() permite especificar a quantidade de tempo que uma *thread* deve esperar para estabelecer um bloqueio antes de retornar verdadeiro ou falso.

A seguir teremos uma classe de implementação para *MyPrinter* (em nosso exemplo de *MyThread2*) usando bloqueios. Observe que não existe nenhum desbloqueio implícito, isso acontece quando o bloco sincronizado termina. Assim, qualquer chamada aos métodos *lock*() ou *tryLock*() deve ser seguida por uma chamada ao método *unlock*().

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

class MyPrinter {
  // ReentrantLock é uma implementação da interface lock
  final Lock l = new ReentrantLock();

  public void print10(int value) {
    boolean gotLock = false;
    while (!gotLock) {
      gotLock = l.tryLock();
      if (!gotLock) {
        System.out.println("Unable to get lock, try again next time");
        try {
          Thread.sleep((int)(Math.random() * 1000));
        } catch (InterruptedException e) { }
      }
    }
    for (int i = 0; i < 10; i++) {
      System.out.print(value);
    }
    System.out.println(""); // Uma nova linha após 10 números
    l.unlock();
  }
}
```

Os métodos *wait*() e *notify*() só podem ser chamados no bloqueio intrínseco de um método ou bloco sincronizado. Para fazer isso, podemos usar seu método *newCondition*() que devolve um objeto do tipo *Condition*. Um objeto *Condition* tem um método denominado *await*() que é o equivalente a chamada ao método *wait*(). Para retornar uma *thread* que está em espera o objeto *Condition* tem um método denominado *signal*() que é o equivalente a uma chamada ao método *notify*().

Por exemplo, adicionaremos as seguintes linhas a classe *SharedVars* usando bloqueios e

Conditions.

```
import java.util.concurrent.locks.*;
class SharedVars {
  final Lock l = new ReentrantLock();
  final Condition emptyCondition = l.newCondition();
  int array[] = new int[100];
  int top;

  // método para inserir um novo valor
  public void insertValue(int value) {
    l.lock() // realiza o bloqueio
    if (top < array.length) {
      array[top] =  value;
      top++;
      if (top == 1) {
        emptyCondition.signal();
      }
    }
    l.unlock();
  }
  // método para obter um valor
  public int getValue() {
    l.lock();
    if (top <= 0) {
      try {
        emptyCondition.await();
      } catch (InterruptedException e) { }
    }
    top--;
    l.unlock();
    return array[top];
  }
}
```

Existem outras classes no pacote *java.util.concurrent.locks*. Uma delas é a classe *ReentrantReadWriteLock* que classifica os métodos que retornam os bloqueios. Estes bloqueios podem ser usados como uma solução ao problema **produtor-consumidor**.

5.2. Executores

Ao criamos uma *thread*, declaramos manualmente quando esta *thread* será executada. Por exemplo, a *thread* seguinte:

```
class MyThread implements Runnable { ... }
```

Manualmente, iniciamos a *thread* através das seguintes instruções:

```
Thread t1 = new Thread(mt1);
t1.start();
```

Uma classe que implementa a interface *Executor* pode iniciar automaticamente uma *thread*. Por exemplo, em vez das instruções anteriores, teríamos:

```
MyThread mt1 = new MyThread();
Executor e = new ThreadPoolExecutor(10); // accepts max 10 threads
e.execute(mt1);
e.execute(otherThreads);
```

Nosso exemplo utiliza uma classe denominada *ThreadPoolExecutor*. Esta classe executa nossas *threads* inseridas (pelo método execute()) através de um pool de *threads* em execução. Estas *threads* em execução reduz a necessidade do *overhead* para a criação de uma *thread*, uma *threads* é passada como um parâmetro para uma linha que já se encontra em execução.

Um parâmetro especificado no construtor indica quantas *threads* podem rodar no *Executor* de

modo simultâneo. Se um *Executor* não for usado, então as *threads* são executadas no momento em que são iniciadas. Se houver muitas *threads* executando simultaneamente, o sistema pode chegar a não suportar e todas as *threads* serão interrompidas. Um *executor* permite que um número de máximo de *threads* em execução seja especificado e permite que o sistema controle este número, criando um agrupamento daquelas que não pode.

Métodos adicionais indicam como novas *threads* devem ser colocadas no pool ou quando terminar as *threads* que estão inativas. Uma subclasse de *ThreadPoolExecutor*, é a ScheduledThreadPoolExecutor que possui métodos para permitir que uma *thread* em particular inicie em um determinado momento, e seja repetida periodicamente.

5.3. Coleções Concorrentes

O pacote *java.util.concurrent* inclui uma coleção de classes que podem ser usadas com programas *multithread*, sem termos que nos preocupar com problemas de sincronização. Por exemplo, a classe de *BlockingQueue* cria uma FIFO (*first in – first out*) que bloqueia automaticamente *threads* que tentam entrar em um *pool* cheio ou retorna valores vazio. A execução retoma automaticamente quando outra *thread* remove um valor em um pool cheio ou acrescenta novos dados a um vazio.

5.4. Variáveis Atômicas

Como vimos no problema **produtor-consumidor**, as instruções:

```
array[top] = <new value>;
top++;
```

Devem ser executadas sem intervalos. Este processo é chamado de execução atômica. Cada linha de instrução deve ser executada como um todo. Porém, nem mesmo únicas instruções são atômicas. Considere a seguinte instrução:

```
top++;
```

Esta instrução está dividida em três partes quando é executada na CPU:

1. Obtém o valor de **top**
2. Adiciona **1** para o valor
3. Salva o valor no endereço de **top**

Estas três instruções devem ser executadas sem deixar que nenhuma outra instrução seja executada. Um modo para assegurarmos isso, é proteger estas instruções por um método sincronizado.

```
class SharedVars {
  int top;
  public synchronized inctop() { top++; }
}
```

A maioria das outras operações aritméticas não são atômicas, isso significa que cada operação matemática deve lidar com variáveis compartilhadas, e tem que estar em um bloco sincronizado para serem seguras. Entretanto, isto pode incluir instruções para tarefas compartilhadas. Para evitar todo este trabalho, o pacote *java.util.concurrent.atomic* fornece classes preparadas para trabalhar de forma segura em ambiente *multithread*, ou seja, são *thread-safe*.

Por exemplo, a classe *AtomicInteger* fornece os seguintes métodos:

- *void incrementAndGet*() – executa atomicamente uma operação de incremento
- *void* addAndGet(int delta) – acrescenta um valor delta a *AtomicInteger*

Também existem implementações, tais como, um *array* de tipos inteiros ou um objeto de referência. A maioria das classes em *java.util* é *thread-safe*.

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.

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

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