# Multithreading em JAVA

Prof. MSc. Daniel Menin Tortelli

# CountDownLatch

Aplicação exemplo: **CountDownLatch**

# CountDownLatch

- Um auxílio de sincronização que permite que uma ou mais threads aguardem até que um conjunto de operações sendo realizadas em outras threads seja concluído.
- Um **CountDownLatch** possui um campo de contador, que pode ser decrementado conforme necessário.
- Pode ser usado para bloquear a execução de uma thread até que o contador seja zero.
- Em um processamento paralelo, é possível instanciar o **CountDownLatch** com o mesmo valor para o contador como um número de threads que irão efetuar alguma tarefa.
- Então, chama-se a função **countdown()** após cada thread terminar, garantindo que um thread dependente que chama **await()** será bloqueado até que TODAS as threads sejam concluídas.

# CountDownLatch

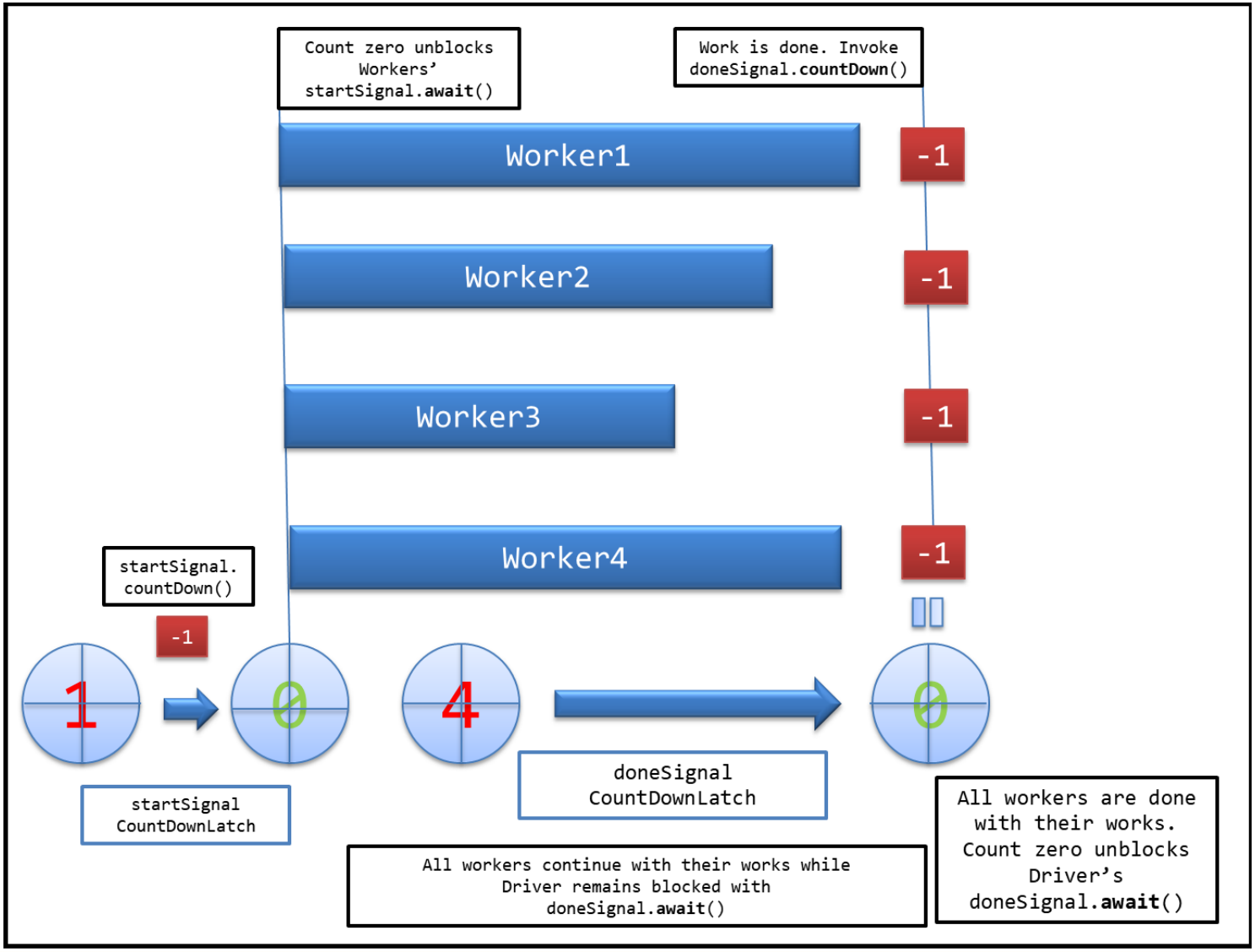
```
public Processor(int _ID, int _sleepTime, CountDownLatch _latch)
{
    this.ID = _ID;
    this.sleepTime = _sleepTime;
    this.latch = _latch;
}

@Override
public void run()
{
    System.out.println("Thread: " + ID + " iniciou... " + this.sleepTime);

    // Executa a tarefa da thread
    try
    {
        Thread.sleep(this.sleepTime);
    }
    catch (InterruptedException ex)
    {
        ex.printStackTrace();
    }

    // Decrementa o contador após concluir a tarefa.
    latch.countDown();

    System.out.println("Thread: " + ID + " - Latch Count: " + latch.getCount());
}
```

# Latch

```
public static void main(String[] args)
{
    Random random = new Random();

    // Cria o CountDownLatch e define o contador para 3
    CountDownLatch latch = new CountDownLatch(3);

    // Cria um pool para a execução simultânea de 3 threads
    ExecutorService executor = Executors.newFixedThreadPool(3);

    // Envia as threads para execução
    for (int i = 0; i < 3; i++)
    {
        int sleepTime = random.nextInt(4) * 1000 + 1000;
        executor.submit(new Processor(i, sleepTime, latch));
    }

    // Encerra as threads
    executor.shutdown();

    // Aguarda até que o contador chegue a ZERO para prosseguir
    // com a execução do programa
    try
    {
        latch.await();
    }
    catch (InterruptedException ex)
    {
        ex.printStackTrace();
    }

    System.out.println("Fim MAIN");
}
```

```
run:
Thread: 0 iniciou... 2000
Thread: 1 iniciou... 3000
Thread: 2 iniciou... 1000
Thread: 2 - Latch Count: 2
Thread: 0 - Latch Count: 1
Thread: 1 - Latch Count: 0
Fim MAIN
BUILD SUCCESSFUL (total time: 3 seconds)
```

# CountDownLatch



# CountDownLatch

```
class Worker implements Runnable
{
    private final CountDownLatch startSignal;
    private final CountDownLatch doneSignal;
    private int ID;
    private Random random = new Random();

    Worker(int _ID, CountDownLatch _startSignal, CountDownLatch _doneSignal)
    {
        this.ID = _ID;
        this.startSignal = _startSignal;
        this.doneSignal = _doneSignal;
    }

    public void run()
    {
        try
        {
            startSignal.await();
            doWork();
            doneSignal.countDown();

            System.out.println("Thread: " + ID + " - Latch Count: " +
                    this.doneSignal.getCount());
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        } // return;
    }
```

# CountDownLatch

```
    void doWork()
    {
        int workTime = random.nextInt(4) * 1000 + 1000;

        System.out.println("Executando tarefa da thread: " + this.ID +
            " com : " + workTime + " milisegundos...");

        try
        {
            Thread.sleep(workTime);
        }
        catch (InterruptedException ex)
        {
            ex.printStackTrace();
        }
    }
}
```

# CountDownLatch

```
public static void main(String[] args)
{
    CountDownLatch startSignal = new CountDownLatch(1);
    CountDownLatch doneSignal = new CountDownLatch(3);

    for (int i = 0; i < 3; ++i) // create and start threads
    {
        new Thread(new Worker(i, startSignal, doneSignal)).start();
    }

    // Executa alguma tarefa
    try
    {
        System.out.println("Executando primeira tarefa na MAIN thread, " +
        "ANTES de iniciar outras threads...");
        Thread.sleep(5000);
    }
    catch (InterruptedException ex)
    {
        ex.printStackTrace();
    }

    // don't let run yet
    startSignal.countDown();      // let all threads proceed
```

# CountDownLatch

```
    // Executa alguma tarefa
    try
    {
        System.out.println("Executando primeira tarefa na MAIN thread, " +
        "DEPOIS de iniciar outras threads...");
        Thread.sleep(5000);
    }
    catch (InterruptedException ex)
    {
        ex.printStackTrace();
    }

    // Aguarda a finalização de TODAS as threads
    try
    {
        doneSignal.await();   // wait for all to finish
    }
    catch (InterruptedException ex)
    {
       ex.printStackTrace();
    }

    System.out.println("FIM da MAIN thread...");
}
```

# CountDownLatch

```
run-single:
Executando primeira tarefa na MAIN thread, ANTES de iniciar outras threads...
Executando primeira tarefa na MAIN thread, DEPOIS de iniciar outras threads...
Executando tarefa da thread: 0 com : 3000 milisegundos...
Executando tarefa da thread: 1 com : 4000 milisegundos...
Executando tarefa da thread: 2 com : 2000 milisegundos...
Thread: 2 - Latch Count: 2
Thread: 0 - Latch Count: 1
Thread: 1 - Latch Count: 0
FIM da MAIN thread...
BUILD SUCCESSFUL (total time: 10 seconds)
```

# Relacionamento entre Produtor/Consumidor *Sem* Sincronização

# Produtor/Consumidor sem sincronização

- Em um relacionamento entre dois processos produtor/consumidor:
  - A parte **produtora** de um aplicativo gera dados e os armazena em um objeto compartilhado (buffer).
  - A parte **consumidora** lê os dados do objeto compartilhado (buffer).
- Exemplos: spooling de impressão, gravação de CD/DVD.

# Produtor/Consumidor sem sincronização

- Em um relacionamento produtor/consumidor de múltiplas threads...
  - Uma **thread produtora** gera dados e os coloca em um objeto compartilhado chamado **buffer**.
  - Uma **thread consumidora** lê dados do buffer.
  - Se a thread ***produtora*** que está esperando para colocar os próximos dados no buffer determinar que a thread ***consumidora*** ainda não leu os dados anteriores, a thread produtora deve chamar ***await***.
  - Assim, a thread consumidora pode ler os dados antes das atualizações adicionais.
  - Quando a thread ***consumidora*** ler os dados, ela deve chamar ***signal*** para permitir que uma thread produtora (em espera) armazene o próximo valor.

# Produtor/Consumidor sem sincronização

- Em um relacionamento produtor/consumidor de múltiplas threads...
  - Se uma thread consumidora localizar o buffer vazio ou achar que todos os dados já foram lidos, ela deve chamar ***await***.
  - Quando uma thread produtora colocar os próximos dados no buffer, ela deve chamar ***signal*** para permitir que a thread consumidora prossiga, lendo os novos dados.
  - ***Se as threads produtora/consumidora não são sincronizadas, os dados podem ser perdidos se a produtora colocar novos dados no buffer compartilhado antes de a consumidora consumir os dados anteriores.***
  - ***Também pode haver duplicação de dados se a consumidora consumir os dados outra vez antes da produtora produzir o próximo valor.***

# Relacionamento entre Produtor/Consumidor Sem Sincronização

Aplicação exemplo: **SharedBufferTest**

# Exemplo 1 (Buffer.java)

```
// Interface Buffer especifica métodos chamados por Producer e Consumer
// Interface Buffer utilizada nos exemplos de produtor/consumidor
public interface Buffer
{
    public void set(int value); // coloca um valor int no buffer
    public int get();           // retorna o valor int a partir do buffer

} // fim da interface Buffer
```

# Exemplo 1 (Producer.java)

```
// A classe Producer representa a thread Produtora em um relacionamento
// Produtor/Consumidor
import java.util.Random;

// O método run de Producer armazena os valores de 1 a 10 no buffer
public class Producer implements Runnable
{
    private static Random generator = new Random(); // gerador de números aleatórios
    private Buffer sharedLocation;  // referência a objeto compartilhado

    // construtor
    public Producer( Buffer shared )
    {
        sharedLocation = shared;
    } // fim do construtor Producer
```

# Exemplo 1 (Producer.java)

```
    // armazena os valores de 1 a 10 em sharedLocation
    @Override
    public void run()
    {
        int sum = 0;

        for (int count = 1; count <= 10; count++)
        {
            try // dorme de 0 a 3 segundos, então coloca valor no buffer
            {
                // a thread dorme... espera sincronizada
                Thread.sleep(generator.nextInt(3000));
                sharedLocation.set(count); // configura o valor no buffer
                sum += count; // incrementa a soma dos valores
                System.out.printf("\t\t%2d\n", sum); // imprime o somatório
            } // fim do try
            // se a thread adormecida é interrompida, imprime rastreamento de pilha
            catch (InterruptedException exception)
            {
                exception.printStackTrace();
            } // fim do catch
        } // fim do for

        System.out.printf("\n%s\n%s\n",
                "Produtor terminou a produção de dados",
                "Fim do Produtor!\n");
    } // fim do método run
} // fim classe Producer
```

# Exemplo 1 (Consumer.java)

```
// A classe Consumer representa a thread consumidora em um relacionamento
// Produtor/Consumidor
// O método run da classe Consumer itera dez vezes lendo um valor do buffer
import java.util.Random;

public class Consumer implements Runnable
{
    private static Random generator = new Random();
    private Buffer sharedLocation; // referência a objeto compartilhado

    // construtor
    public Consumer( Buffer shared )
    {
        sharedLocation = shared;
    } // fim do construtor Producer
```

# Exemplo 1 (Consumer.java)

```
// lê o valor de sharedLocation quatro vezes e soma os valores
@Override
public void run()
{
    int sum = 0;

    for (int count = 1; count <= 10; count++)
    {
        try // dorme de 0 a 3 segundos, então lê valor no buffer
        {
            // a thread dorme... espera sincronizada
            Thread.sleep(generator.nextInt(3000));
            sum += sharedLocation.get(); // adiciona valor lido à soma
            System.out.printf("\t\t\t\t%2d\n", sum); // imprime o somatório
        } // fim do try
        // se a thread adormecida é interrompida, imprime rastreamento de pilha
        catch (InterruptedException exception)
        {
            exception.printStackTrace();
        } // fim do catch
    } // fim do for

    System.out.printf("\n%s %d.\n%s\n",
            "A soma dos valores lidos pelo Consumidor foi ",
            sum,
            "Fim do Consumidor!\n");
} // fim método run
} // fim da classe Consumer
```

# Exemplo 1 (UnsynchronizedBuffer.java)

```
// UnsynchronizedBuffer representa um único inteiro compartilhado
// UnsynchronizedBuffer mantém o inteiro compartilhado que é acessado
// por uma thread produtora e uma consumidora por meio dos métodos
// set e get
public class UnsynchronizedBuffer implements Buffer
{
    // compartilhado pelas threads Producer e Consumer
    private int buffer = -1;

    // Coloca o valor no buffer
    @Override
    public void set(int value)
    {
        System.out.printf("Produtor escreve\t %2d", value);
        buffer = value;
    } // fim do método set

    // Retorna o valor do buffer
    @Override
    public int get()
    {
        System.out.printf("Consumidor lê\t\t %2d", buffer);
        return buffer;
    } // fim do método get
} // fim da classe UnsynchronizedBuffer
```

# Exemplo 1 (SharedBufferTest.java)

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest
{
    public static void main(String[] args)
    {
        // cria novo pool de threads com duas threads
        ExecutorService application = Executors.newFixedThreadPool( 2 );

        // cria UnsynchronizedBuffer para armazenar inteiros
        Buffer sharedLocation = new UnsynchronizedBuffer();

        System.out.println("Ação\t\t\tValor\t     Produziu\t     Consumiu");
        System.out.println("----------------\t-----\t     --------\t     --------");

        // Tenta iniciar as threads produtora e consumidora
        // fornecendo acesso a cada uma a sharedLocation
        try
        {
            application.execute(new Producer(sharedLocation));
            application.execute(new Consumer(sharedLocation));
        } // fim do try
        catch (Exception exception)
        {
            exception.printStackTrace();;
        } // fim do catch

        application.shutdown(); // termina aplicativo quando as threads terminam

    } // fim do main
} // fim da classe SharedBufferTest
```

# Exemplo 1 (Resultados)

| Ação | Valor | Produziu | Consumiu |
|---|---|---|---|
| Produtor escreve | 1 | 1 | |
| Consumidor lê | 1 | | 1 |
| Consumidor lê | 1 | | 2 |
| Produtor escreve | 2 | 3 | |
| Produtor escreve | 3 | 6 | |
| Produtor escreve | 4 | 10 | |
| Produtor escreve | 5 | 15 | |
| Produtor escreve | 6 | 21 | |
| Consumidor lê | 6 | | 8 |
| Consumidor lê | 6 | | 14 |
| Produtor escreve | 7 | 28 | |
| Consumidor lê | 7 | | 21 |
| Produtor escreve | 8 | 36 | |
| Produtor escreve | 9 | 45 | |
| Consumidor lê | 9 | | 30 |
| Consumidor lê | 9 | | 39 |
| Consumidor lê | 9 | | 48 |
| Produtor escreve | 10 | 55 | |

Produtor terminou a produção de dados
Fim do Produtor!

| | | | |
|---|---|---|---|
| Consumidor lê | 10 | | 58 |
| Consumidor lê | 10 | | 68 |

A soma dos valores lidos pelo Consumidor foi 68.
Fim do Consumidor!

CONSTRUÍDO COM SUCESSO (tempo total: 15 segundos)

| Ação | Valor | Produziu | Consumiu |
|---|---|---|---|
| Consumidor lê | -1 | | -1 |
| Produtor escreve | 1 | 1 | |
| Consumidor lê | 1 | | 0 |
| Produtor escreve | 2 | 3 | |
| Consumidor lê | 2 | | 2 |
| Consumidor lê | 2 | | 4 |
| Consumidor lê | 2 | | 6 |
| Produtor escreve | 3 | 6 | |
| Consumidor lê | 3 | | 9 |
| Consumidor lê | 3 | | 12 |
| Produtor escreve | 4 | 10 | |
| Consumidor lê | 4 | | 16 |
| Produtor escreve | 5 | 15 | |
| Consumidor lê | 5 | | 21 |
| Consumidor lê | 5 | | 26 |

A soma dos valores lidos pelo Consumidor foi 26.
Fim do Consumidor!

| | | | |
|---|---|---|---|
| Produtor escreve | 6 | 21 | |
| Produtor escreve | 7 | 28 | |
| Produtor escreve | 8 | 36 | |
| Produtor escreve | 9 | 45 | |
| Produtor escreve | 10 | 55 | |

Produtor terminou a produção de dados
Fim do Produtor!

CONSTRUÍDO COM SUCESSO (tempo total: 20 segundos)

# Métodos da Classe Thread

# Métodos da Classe Thread

| Método | Descrição |
|---|---|
| **activeCount()** | Retorna o número de threads ativas |
| **currentThread()** | Retorna a referência da thread que está executando |
| | |
| **setName(nome)** | Define um nome para a thread |
| **getName()** | Obtém o nome da thread |
| | |
| **setPriority(int)** | define a prioridade da thread.<br>MAX_PRIORITY, NORM_PRIORITY, MIN_PRIORITY |
| **getPriority()** | Obtém a prioridade da thread |

# Métodos da Classe Thread

| Método | Descrição |
|---|---|
| **join() , join(millis)** | A thread atual chama esse método para uma segunda thread, fazendo com que a thread atual permaneça bloqueada até que a segunda thread termine sua tarefa. |
| **sleep(millis)** | Determina que uma thread irá dormir por um determinando tempo (Espera Sincronizada) |
| | |
| **yield()** | Causa uma pausa temporária (obsoleto) |
| **interrupt()** | Interrompe uma thread (se estiver bloqueada ou dormindo) |
| **suspend()** | Provoca uma pausa na thread |
| **resume()** | Desbloqueia uma thread que está no modo pausa |
| **stop()** | Finaliza uma thread |

# Relacionamento entre Produtor/Consumidor *Com* Sincronização

Aplicação exemplo: **SharedBufferTestSync**

# Produtor/Consumidor com Sincronização

- Nesse caso, a consumidora consome corretamente um valor apenas depois de a produtora ter produzido um valor.

- A produtora, por sua vez, produz corretamente um novo valor apenas depois de a consumidora ter consumido o valor produzido anteriormente.

- Essa abordagem permite demonstrar que as threads que acessam o objeto compartilhado (buffer) não estão cientes de que estão sendo sincronizadas.

# Exemplo 2 (Buffer.java)

```
// Interface Buffer especifica métodos chamados por Producer e Consumer
// Interface Buffer utilizada nos exemplos de produtor/consumidor
public interface Buffer
{
    public void set(int value); // coloca um valor int no buffer
    public int get();           // retorna o valor int a partir do buffer

} // fim da interface Buffer
```

# Exemplo 2 (Producer.java)

```
// O método run de Producer armazena os valores de 1 a 10 no buffer
public class Producer implements Runnable
{
    private static Random generator = new Random(); // gerador de números aleatórios
    private Buffer sharedLocation;  // referência a objeto compartilhado

    // construtor
    public Producer( Buffer shared )
    {
        sharedLocation = shared;
    } // fim do construtor Producer
```

# Exemplo 2 (Producer.java)

```
// armazena os valores de 1 a 10 em sharedLocation
@Override
public void run()
{
    int sum = 0;

    for (int count = 1; count <= 10; count++)
    {
        try // dorme de 0 a 3 segundos, então coloca valor no buffer
        {
            // a thread dorme... espera sincronizada
            Thread.sleep(generator.nextInt(3000));
            sharedLocation.set(count); // configura o valor no buffer
            sum += count; // incrementa a soma dos valores
            //System.out.printf("\t\t%2d\n", sum); // imprime o somatório
        } // fim do try
        // se a thread adormecida é interrompida, imprime rastreamento de pilha
        catch (InterruptedException exception)
        {
            exception.printStackTrace();
        } // fim do catch
    } // fim do for

    System.out.printf("\n%s\n%s\n",
            "Produtor terminou a produção de dados",
            "Fim do Produtor!\n");
} // fim do método run
} // fim classe Producer
```

# Exemplo 2 (Consumer.java)

```
public class Consumer implements Runnable
{
    private static Random generator = new Random();
    private Buffer sharedLocation; // referência a objeto compartilhado

    // construtor
    public Consumer( Buffer shared )
    {
        sharedLocation = shared;
    } // fim do construtor Producer
```

# Exemplo 2 (Consumer.java)

```
// lê o valor de sharedLocation quatro vezes e soma os valores
@Override
public void run()
{
    int sum = 0;

    for (int count = 1; count <= 10; count++)
    {
        try // dorme de 0 a 3 segundos, então lê valor no buffer
        {
            // a thread dorme... espera sincronizada
            Thread.sleep(generator.nextInt(3000));
            sum += sharedLocation.get(); // adiciona valor lido à soma
            //System.out.printf("\t\t\t\t%2d\n", sum); // imprime o somatório
        } // fim do try
        // se a thread adormecida é interrompida, imprime rastreamento de pilha
        catch (InterruptedException exception)
        {
            exception.printStackTrace();
        } // fim do catch
    } // fim do for

    System.out.printf("\n%s %d.\n%s\n",
            "A soma dos valores lidos pelo Consumidor foi ",
            sum,
            "Fim do Consumidor!\n");
} // fim método run
} // fim da classe Consumer
```

# Exemplo 2 (SynchronizedBuffer)

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

// SynchronizedBuffer sincroniza acesso a um único inteiro compartilhado
public class SynchronizedBuffer implements Buffer
{
    // Bloqueio para controlar a sincronização com esse buffer
    private Lock accessLock = new ReentrantLock();

    // Condições para controlar a leitura e gravação
    private Condition canWrite = accessLock.newCondition();
    private Condition canRead = accessLock.newCondition();

    // compartilhado pelas threads Producer e Consumer
    private int buffer = -1;
    private boolean occupied = false;   // se o buffer estiver ocupado...
```

# Exemplo 2 (SynchronizedBuffer)

```
// Coloca o valor inteiro no buffer
@Override
public void set(int value)
{
    accessLock.lock();  // bloqueia esse objeto

    // envia informações de thread e de buffer para a saída, então espera
    try
    {
        while ( occupied == true )
        {
            System.out.println("Produtora tentando escrever no buffer");
            displayState("Buffer cheio. Produtora aguarda...");
            canWrite.await();   // espera até que o buffer esteja vazio
        } // fim do while

        buffer = value; // configura novo valor de buffer

        // indica que a produtora não pode armazenar outro valor
        // até a consumidora recuperar valor atual de buffer
        occupied = true;

        displayState("Produtora escreve " + buffer);

        // sinaliza a thread que está esperando para ler a partir do buffer
        canRead.signal();
    } // fim do try
    catch (InterruptedException exception)
    {
        exception.printStackTrace();
    } // fim do catch
    finally
    {
        accessLock.unlock(); // desbloqueia esse objeto
    } // fim do finally
} // fim do método set
```

# Exemplo 2 (SynchronizedBuffer)

```java
// Retorna o valor do buffer
@Override
public int get()
{
    int readValue = 0; // inicializa valor lido a partir do buffer
    accessLock.lock(); // bloqueia esse objeto

    // envia informações  de thread e de buffer para a saída, então espera
    try
    {
        // enquanto os dados não são lidos, coloca a thread em estado de espera
        while ( !occupied )
        {
            System.out.println("Consumidora tenta ler do buffer");
            displayState("Buffer vazio. Consumidora aguarda...");
            canRead.await(); // espera até o buffer tornar-se cheio
        } // fim do while

        // indica que a produtora pode armazenar outro valor
        // porque a consumidora acabou de recuperar o valor do buffer
        occupied = false;

        readValue = buffer; // recupera o valor do buffer
        displayState("Consumidora lê " + readValue);

        // sinaliza a thread que está esperando o buffer tornar-se vazio
        canWrite.signal();
    } // fim do try
    // se a thread na espera estiver sido interrompida, imprime o
    // rastreamento da pilha
    catch (Exception exception)
    {
        exception.printStackTrace();
    } // fim do catch
    finally
    {
        accessLock.unlock(); // desbloqueia o objeto
    } // fim do finally

    return readValue;
} // fim do método get
```

# Exemplo 2 (SynchronizedBuffer)

```
    public void displayState(String operation)
    {
        System.out.printf("%-40s%d\t\t%b\n\n", operation, buffer, occupied);
    } // fim do método displayState

} // fim da classe SynchronizedBuffer
```

# Exemplo 2 (SharedBufferTestSync)

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;


public class SharedBufferTestSync
{

    public static void main(String[] args)
    {
       // cria novo pool de threads com duas threads
        ExecutorService application = Executors.newFixedThreadPool( 2 );

        // cria SynchronizedBuffer par armazenar inteiros
        Buffer sharedLocation = new SynchronizedBuffer();

        System.out.printf("%-40s%s\t\t%s\n%-40s%s\n\n", "Operação",
                "Buffer", "Ocupado", "----------", "-------\t\t---------");

        // tenta iniciar a produtora e a consumidora
        try
        {
            application.execute(new Producer(sharedLocation));
            application.execute(new Consumer(sharedLocation));
        } // fim do try
        catch (Exception exception)
        {
            exception.printStackTrace();
        } // fim do catch

        application.shutdown();

    } // fim do main
} // fim da classe SharedBufferTestSync
```

| Operação | Buffer | Ocupado |
|---|---|---|
| Produtora escreve 1 | 1 | true |
| Consumidora lê 1 | 1 | false |
| Produtora escreve 2 | 2 | true |
| Consumidora lê 2 | 2 | false |
| Produtora escreve 3 | 3 | true |
| Produtora tentando escrever no buffer<br>Buffer cheio. Produtora aguarda... | 3 | true |
| Consumidora lê 3 | 3 | false |
| Produtora escreve 4 | 4 | true |
| Consumidora lê 4 | 4 | false |
| Consumidora tenta ler do buffer<br>Buffer vazio. Consumidora aguarda... | 4 | false |
| Produtora escreve 5 | 5 | true |
| Consumidora lê 5 | 5 | false |
| Produtora escreve 6 | 6 | true |
| Produtora tentando escrever no buffer<br>Buffer cheio. Produtora aguarda... | 6 | true |
| Consumidora lê 6 | 6 | false |
| Produtora escreve 7 | 7 | true |
| Consumidora lê 7 | 7 | false |
| Consumidora tenta ler do buffer<br>Buffer vazio. Consumidora aguarda... | 7 | false |
| Produtora escreve 8 | 8 | true |
| Consumidora lê 8 | 8 | false |
| Produtora escreve 9 | 9 | true |
| Consumidora lê 9 | 9 | false |
| Consumidora tenta ler do buffer<br>Buffer vazio. Consumidora aguarda... | 9 | false |
| Produtora escreve 10 | 10 | true |
| Consumidora lê 10 | 10 | false |

A soma dos valores lidos pelo Consumidor foi 55.
Fim do Consumidor!

Produtor terminou a produção de dados
Fim do Produtor!

# Sincronização de Threads

# Sincronização de Threads

- Frequentemente, múltiplas threads em execução manipulam um objeto compartilhado na memória.
- Pode haver problemas de consistência nos resultados se este objeto for modificado por uma ou mais threads.
- É necessário que o objeto compartilhado seja gerenciado.
- (Thread atualizando objeto e outra thread tentando chamá-lo.)

# Sincronização de Threads

- Esse problema pode ser resolvido fornecendo a uma thread por vez o **acesso exclusivo** ao objeto compartilhado (buffer).
- Enquanto uma thread tem o controle do objeto, as outras threads ficam em modo de espera.
- Quando a thread com o controle do objeto terminar de usá-lo, outra thread na fila recebe o controle do mesmo.
- ***Assim, toda a thread que acessa o objeto compartilhado exclui todas as outras threads de fazer isso simultaneamente.***
- Isso é chamado de **Exclusão Mútua**.

# Sincronização de Threads

- ***A exclusão mútua permite ao programador realizar a sincronização de threads.***

- Ela coordena o acesso aos dados compartilhados por múltiplas threads concorrentes.

- O JAVA utiliza **bloqueios** para a realização da sincronização.

# Sincronização de Threads

- Qualquer objeto pode conter um objeto que implementa a interface **Lock** (*pacote java.util.concurrent.locks*).
- Uma thread chama o método **lock** da interface Lock para obter o bloqueio (acesso exclusivo).
- Uma vez que o Lock foi obtido por uma thread, o objeto Lock não permitirá que outra thread obtenha o bloqueio.
- Isso só será possível se a thread que tem o objeto libere o Lock (através do método **unlock** da interface Lock).

# Sincronização de Threads

- Quando uma thread chamar o método **unlock**, o bloqueio no objeto será liberado.

- ***A thread na fila de espera que possuir a maior prioridade poderá bloquear o objeto para si.***

- Utiliza-se a classe **ReentrantLock** (*pacote java.util.concurrent.locks*) para implementação básica da interface Lock.

# Sincronização de Threads

- O construtor de **ReentrantLock** possui um argumento *booleano*.

- Se for passado **true**, a thread com a espera mais longa vai poder bloquear o objeto.

- Se for **false**, não se sabe qual thread na fila de espera vai receber o controle do objeto.

# Sincronização de Threads

- Se uma thread que possui o bloqueio em um objeto determinar que não é possível continuar sua tarefa até que alguma condição seja satisfeita, a thread pode esperar em uma **variável de condição**.

- Isso dispensa a thread da disputa pelo processador, coloca-a em uma fila de espera pela variável de condição e libera o bloqueio no objeto.

- As variáveis de condição devem ser associadas com um Lock e são criadas chamando o método Lock **newCondition**, que retorna um objeto que implementa a interface **Condition** (pacote java.util.concurrent.locks).

# Sincronização de Threads

- Para esperar uma variável de condição, a thread pode chamar o método **await** de *Condition*.

- Isso libera imediatamente o Lock associado e coloca a thread no estado de espera dessa *Condition*.

- Outras threads podem então tentar obter o Lock.

# Sincronização de Threads

- Quando a thread executável completar a tarefa e determinar que a thread na espera pode agora continuar, a thread executável pode chamar o método **signal** da interface *Condition*.

- Se uma thread chamar o método **signalAll**, todas as threads que esperam essa condição mudam para o estado executável e podem requisitar o Lock.

- Apenas uma pode obter o Lock no objeto por vez.

# Impasse (Deadlock)

- Um impasse (deadlock) ocorre quando uma thread (thread1) não pode prosseguir porque está esperando (direta ou indiretamente) outra thread (thread2) prosseguir.

- Simultaneamente, a thread2 não pode prosseguir porque está esperando a thread1 prosseguir.

- Como as duas threads estão esperando uma à outra, as ações que permitiriam a cada uma continuar a execução nunca ocorrem.

# Impasse (Deadlock)

- O bloqueio que ocorre com a execução dos métodos **lock** e **unlock** poderia levar a um impasse se os bloqueios nunca fossem liberados.

- As chamadas para o método **unlock** devem ser colocadas em blocos **finally** para assegurar que os bloqueios sejam liberados e evitar esses tipos de deadlocks.

# Dicas Importantes

- Coloque as chamadas para o método **Lock unlock** em um bloco *finally*.

- Se uma exceção for lançada, o desbloqueio ainda deve ser chamado ou o impasse (deadlock) pode ocorrer.

# Dicas Importantes

- Sempre invoque o método ***await*** em um loop que testa uma condição apropriada.

- É possível que uma thread entre novamente no estado executável antes que a condição que ela estava esperando seja satisfeita.

- Testar a condição novamente assegura que a thread não executará de maneira errada se ela tiver sido sinalizada anteriormente.

# Dicas Importantes

- Esquecer de sinalizar (***signal***) uma thread que está esperando por uma condição é um erro de lógica.

- A thread permanecerá no estado de espera, o que a impedirá de continuar trabalhando.

- Tal espera pode levar ao adiamento indefinido (starvation) ou impasse (deadlock).

# Exercício 1

- Modifique a aplicação **SharedBufferTestSync** para que a thread Produtora:
  - Execute 5 vezes.
  - A cada execução, ela deve gerar um número aleatório de 1 a 10 que será armazenado no buffer.
  - Se o valor do contador do laço de repetição for **Par**, o valor armazenado deve ser **Negativo**.
  - Se o valor do contador do laço de repetição for **Ímpar**, o valor armazenado deve ser **Positivo**.

# Exercício 1

- Modifique a aplicação **SharedBufferTestSync** para que a thread Consumidora:
  - Execute 5 vezes.
  - A cada execução, ela deve retirar o valor do buffer (se ele não estiver vazio) e, logo após:
    - Analisar se o valor for Positivo ou Negativo e Par ou Ímpar.
    - Em todos os casos, devem ser feitos os somatórios da quantidade de valores e dos próprios valores gerados.

# Exercício 2

- Modifique a aplicação **SharedBufferTestSync** para que a thread Produtora:
  - Gere números aleatórios indefinidamente no intervalo de 0 – 100.
  - Armazene apenas **números primos** no buffer.
  - Quando **5** números primos forem gerados, a thread produtora deve encerrar sua tarefa.

# Exercício 2

- Modifique a aplicação **SharedBufferTestSync** para que a thread Consumidora:
  - Recupere os números primos do buffer (a medida que eles forem gerados).
  - Organize e mostre os 5 números primos em **ordem crescente** no final da execução da aplicação.
  - Mostre também um somatório dos números primos encontrados.

# Exercício 3

- Elabore um programa que crie **n** threads. A quantidade de threads criadas deve ser inserida pelo usuário no início do programa.

- Para cada thread criada, deve ser atribuída uma **prioridade** específica de execução. As prioridades vão de **1** a **10** e devem ser geradas aleatoriamente para cada thread no momento de sua criação.

- A única função da thread é imprimir o seu nome, a hora que executou e a sua prioridade.

# Exercício 4

- Elabore um programa que crie **3** threads e atribua um nome a cada uma.

- Em seguida, sorteie uma sequência de números de 1 a 3 que equivale a ordem de execução de cada thread.

- ***Pesquise uma forma de fazer com que as threads executem na ordem sorteada. Ou seja, a segunda thread começará a executar apenas quando a primeira terminar sua execução.***

# Exercício 4

- A **Thread 1** deve imprimir os números de 1 a 5 a cada 1 segundo.
- A **Thread 2** deve imprimir os números de 10 a 5 a cada 1 segundo.
- A **Thread 3** deve imprimir os números de pares de 20 a 50 a cada 1 segundo.

# Exercício 5

- Elabore um programa que crie **2** threads.
- O programa deve simular os procedimentos de crédito e débito em uma conta corrente.
- Utilize a sincronização pois as threads de crédito e débito não devem alterar o valor da conta corrente ao mesmo tempo para realizar suas operações.
- Os valores a serem creditados e debitados devem ser gerados aleatoriamente durante 5 tentativas de creditar e 5 para debitar a conta.
- A conta não pode ficar com saldo negativo. Se ficar, não poderá ser efetuado o débito. Exiba uma mensagem na tela indicando que a conta está negativa se houver uma tentativa de débito indevido.