UNIVERSIDADE DO ESTADO DE SANTA CATARINA
DCC – Departamento de Ciência da Computação
Joinville – SC

CURSO: Bacharelado em Ciência da Computação
DISCIPLINA: POO0001 – Programação Orientada a Objetos
PROFESSOR: Diego Buchinger
AULA 08 – Programação *multithreading* e Coleções (*Collections*) em Java

1. PROGRAMAÇÃO MULTITHREADING

Até um determinado período, a velocidade de processamento nos computadores era definida pela velocidade dos seus recursos, sendo o processador um dos principais componentes envolvidos nesta questão. O aumento do *clock* e da velocidade dos processadores sem dúvidas aumentou muito nos nas últimas décadas, mas hoje já encontra certas problemáticas físicas que permitam uma evolução significativa neste sentido. Devido a esta dificuldade, começou-se a buscar outras formas de melhorar o desempenho e uma delas é baseada na utilização de mais processadores ou mais núcleos de processamento, de tal forma que o processamento não fique centralizado.

Esta nova forma de arquitetura permite uma melhora significativa no tempo de resposta dos computadores, pois tarefas distintas podem ser realizadas em paralelo. Entretanto, para que uma aplicação possa usufruir do benefício de utilizar mais de um núcleo de processamento, devemos programa-la de forma explícita. Nesta seção vamos discutir como podemos fazer isto com o Java.

1.1. Threads

Um thread, ou uma thread, pode ser definido como um fluxo de execução que realiza os comandos definidos em um programa. Cada thread possui recursos próprios, mas também pode compartilhar recursos com outros threads que sejam do mesmo processo. Para exemplificar, considere o coletor de lixo do Java; enquanto o seu programa está executando, existe um thread executando possivelmente em paralelo ao seu programa que está constantemente verificando se existe uma área de memória que deixou de ser utilizada. Ou então um servidor web que pode receber milhares de requisições de conteúdo em alguns segundos; ao invés de colocar cada requisição em uma fila de espera e executar uma por vez, é comum que múltiplos threads recebam requisições de forma independente e retornem um resultado. Outro exemplo bem comum são os editores de texto que, ao mesmo tempo em que estão aceitando a entrada do usuário, estão exibindo o conteúdo na tela de forma gráfica e também estão realizando a correção ortográfica do texto.

O conceito de threads é tratado e estudado com maior detalhamento na disciplina de sistemas operacionais mas, como vamos trabalhar com threads é importante saber que eles podem estar em três estados possíveis: bloqueados (esperando alguma informação de um periférico), prontos (estão prontos para executar, desde que um núcleo de processamento esteja disponível) e em execução (um núcleo de processamento o está executando). A Figura 1 este ciclo de vida dos threads. É importante saber ainda, que na maioria dos sistemas operacionais modernos, um thread não executa de forma ininterrupta; muito pelo contrário, os threads são constantemente parados e trocados/chaveados, a fim de oferecer a sensação de que todos os processos estão produzindo uma resposta na medida em que o tempo avança.

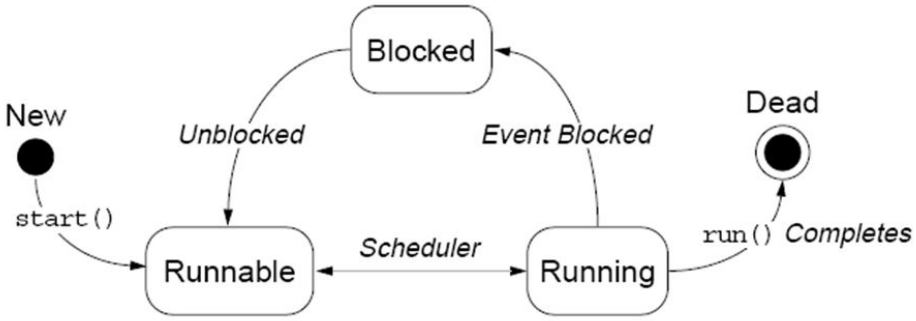

FIGURA 1 – Ciclo de Vida de Threads

1.1. Usando Threads em Java

A linguagem de programação Java permite o uso de threads de maneira nativa, de duas formas distintas: através da classe Thread ou através da interface Runnable. Além da óbvia diferença na sintaxe entre estender uma classe com herança ou implementar uma interface, deve-se considerar que: estender a classe Thread é mais simples e direto, contudo, como Java não permite herança múltipla, essa abordagem só pode ser utilizada se a classe que está estendendo Thread, não herdará de mais alguma outra classe. Caso contrário, o uso da interface Runnable será a única solução viável, uma vez que é possível implementar múltiplas interfaces.

Para fazer uso da classe Thread é necessário estendê-la através de herança. Uma classe derivada de Thread poderá reescrever o método *run()* que é o método onde os threads iniciam a sua execução. Como o método *run()* deve ser reescrito sem parâmetros, é necessário que as inicializações de parâmetros para a execução de um thread sejam feitas no momento da criação do objeto da classe derivada ou pelo menos antes do início da execução do thread. Para iniciar a execução de um thread deve-se ter um cuidado adicional! Isto porque não devemos chamar o método *run()* que foi rescrito, mas sim o método *start()*.

Note que o método *main()* também é executado por um thread, que no caso é o thread principal. Logo, após lançar/inicializar os *threads()*, a execução do método main segue em paralelo. Caso seja de interesse aguardar pelos threads lançados, é possível utilizar o método *join()* do objeto dos threads que estão em execução. O método *join()* bloqueia a execução do thread que o invocou, desbloqueando somente após o término do thread do objeto terminar sua execução ou através de uma exceção de interrupção do thread (InterruptedException), sendo que esta última deve ser tratada ou arremessada adiante. O quadro a seguir mostra uma implementação de uma classe contadora, que possui um nome, um valor de início e um valor de término da contagem. São criados então três objetos contadores e inicializados com valores distintos. Os três são lançados como novos threads e aguarda-se a sua finalização para imprimir uma mensagem final.

De forma ligeiramente diferente, para fazer uso da interface Runnable, é necessário implementá-la, o que obriga a reescrita do método *run()*. O uso de um thread baseado em Runnable, entretanto, é ligeiramente diferente. Deve-se primeiramente instanciar um objeto A de uma classe que implementa esta interface. Na sequencia, deve-se criar um objeto da classe Thread, passando como argumento para o construtor este primeiro objeto A, e somente então iniciar o thread usando o método *start()* deste segundo objeto. O quadro a seguir mostra uma implementação análoga a que foi feita usando herança com a classe Thread.

Implementação Usando a Classe Threads

Arquivo: Contador.java

```
 1.  package multithreading;
 2.
 3.  public class Contador extends Thread{
 4.      int inicio, fim;
 5.      String nome;
 6.
 7.      public Contador(String nome, int inicio, int fim){
 8.          this.nome = nome;
 9.          this.inicio = inicio;
10.          this.fim = fim;
11.      }
12.
13.      @Override
14.      public void run(){
15.          for(int i=inicio; i<=fim; i++)
16.              System.out.println( nome + ": " + i );
17.      }
18.  }
```

Arquivo: Main.java

```
 1.  package multithreading;
 2.
 3.  public class Main {
 4.      public static void main(String[] args){
 5.          Contador ca = new Contador ("Alice",  1, 50);
 6.          Contador cb = new Contador ("Bob",   51, 100);
 7.          Contador cc = new Contador ("Carl", 101, 150);
 8.
 9.          ca.start();
10.          cb.start();
11.          cc.start();
12.
13.          try{
14.              ca.join();
15.              cb.join();
16.              cc.join();
17.          } catch(InterruptedException ex){
18.              System.err.println( ex.getMessage() );
19.          }
20.
21.          System.out.println("BYE BYE BUD");
22.      }
23.  }
```

Implementação Usando a Interface Runnable

Arquivo: Contador.java

```
 1.  package multithreading;
 2.
 3.  public class Contador implements Runnable{
 4.      int inicio, fim;
 5.      String nome;
 6.
 7.      public Contador(String nome, int inicio, int fim){
 8.          this.nome = nome;
 9.          this.inicio = inicio;
10.          this.fim = fim;
11.      }
12.
13.      @Override
14.      public void run(){
15.          for(int i=inicio; i<=fim; i++)
16.              System.out.println( nome + ": " + i );
17.      }
18.  }
```

Arquivo: Main.java

```
 1.  package multithreading;
 2.
 3.  public class Main {
 4.      public static void main(String[] args){
 5.          Contador ca = new Contador ("Alice",  1, 50);
 6.          Contador cb = new Contador ("Bob",   51, 100);
 7.          Contador cc = new Contador ("Carl", 101, 150);
 8.
 9.          Thread ta = new Thread( ca );
10.          Thread tb = new Thread( cb );
11.          Thread tc = new Thread( cc );
12.
13.          ta.start();
14.          tb.start();
15.          tc.start();
16.
17.          try{
18.              ta.join();
19.              tb.join();
20.              tc.join();
21.          } catch(InterruptedException ex){
22.              System.err.println( ex.getMessage() );
23.          }
24.
25.          System.out.println("BYE BYE BUD");
26.      }
27.  }
```

1.2. Problemas no Uso de Threads

O uso de threads é recomendado sempre que possível e desde que seja proveitoso. Em algumas situações, no entanto, deve-se ter atenção redobrada. Quando dois ou mais threads agem sobre um mesmo objeto é possível que ocorra inconsistência de resultado, pois uma linha de código não é atômica, isto é, não necessariamente ocorre de uma só vez sem passos intermediários. E é exatamente neste ponto que pode ocorrer problemas ao modificar um objeto "simultaneamente" com dois ou mais threads. Veja a implementação a seguir, na qual temos um objeto inteiro compartilhado que representa um saldo e dois threads principais: um produtor, que vai inserindo valor ao saldo, e outro consumidor, que vai reduzindo o valor do saldo. Para evidenciar o problema, vamos utilizar uma mesma quantidade de aumento e diminuição do saldo, de tal forma que o valor final esperado seja zero – todo o valor que for somado por um thread será reduzido pelo outro.

Arquivo: Conta.java

```
 1.  package multithreading;
 2.  import java.util.Random;
 3.
 4.  public class Conta {
 5.      static Integer saldo = 0;
 6.      static final int N_VALORES = 500;
 7.
 8.      public static void main(String[] args){
 9.          Producer prod = new Producer();
10.          Consumer cons = new Consumer();
11.          prod.start();
12.          cons.start();
13.          try{
14.              prod.join();
15.              cons.join();
16.          } catch(InterruptedException ex){
17.              System.err.println( ex.getMessage() );
18.          } finally{
19.              System.out.println("Saldo: "+saldo);
20.          }
21.      }
22.  }
23.
24.  class Producer extends Thread {
25.      @Override
26.      public void run(){
27.          for( int i=0; i<Conta.N_VALORES; i++ )
28.              Conta.saldo += 1;
29.      }
30.  }
31.
32.  class Consumer extends Thread {
33.      @Override
34.      public void run(){
35.          for( int i=0; i<Conta.N_VALORES; i++ )
36.              Conta.saldo -= 1;
37.      }
38.  }
```

Ao executar o programa acima, entretanto, “podemos” ter um resultado diferente de zero. A palavra podemos foi deixada entre aspas para evidenciar o fato de que os threads podem efetivamente se interferir ou não. Quanto mais threads e operações sobre um elemento compartilhado, maior será a probabilidade de um erro ocorrer.

1.3. Sincronização entre Threads

Ok, correu um problema de inconsistência, mas podemos fazer algo para evitar tal problema? Evitar que dois threads tentem modificar um mesmo valor ao mesmo tempo? Sim, podemos. Para isto temos pelo menos dois dispositivos de sincronização: uso de métodos sincronizados e o uso de blocos sincronizados baseados em travas.

A primeira opção de sincronização mencionada – uso de métodos sincronizados – permite que possamos definir que um determinado método deve ser sincronizado e não pode executar de forma concorrente com outro método sincronizado. Para tanto, deve-se utilizar a palavra-chave *synchronized* antes do tipo de retorno do método. O que ocorre é que um método sincronizado ativa uma trava implícita dentro da classe que garante que nenhum outro método (dentro da classe) possa destravá-la e executar o seu bloco de comandos, a não ser ele mesmo. O problema de usar métodos sincronizados é que podemos ter situações em que dois atributos distintos estão sendo utilizados por diversos threads e não existe necessidade de proibir a execução de threads que operam sobre atributos diferentes. Nesse sentido, a segundo abordagem é melhor.

Falando da segundo abordagem, ela utiliza regiões de sincronismo que são travadas com base em objetos de trava como referência. Para fazer isto utilizamos uma especificação de bloco no estilo: `synchronized( objeto ){ ... }`. O objeto passado como argumento é o objeto a ser considerado como trava para a execução do bloco. Desta forma é possível distinguir regiões críticas que trabalham com diferentes atributos. Lembre-se que idealmente o valor de um atributo deve ser controlado dentro da própria classe por estas abordagens de sincronismo. O quadro a seguir mostra o uso das duas técnicas, omitindo a parte do código que já foi apresentada anteriormente.

Arquivo: Conta.java

```
 1.  public class Conta {
 2.      private static final Object lock = new Object();
 3.      private static Integer saldo = 0;
 4.      public static synchronized void incrementa(){
 5.          saldo += 1;
 6.      }
 7.      public static  synchronized void decrementa(){
 8.          saldo -= 1;
 9.      }
10.      public static void aumenta(){
11.          synchronized(Conta.lock){
12.              saldo += 1;
13.          }
14.      }
15.      public static void diminui(){
16.          synchronized(Conta.lock){
17.              saldo -= 1;
18.          }
19.      }
20.  }
```

```
21.  class Producer extends Thread {
22.      @Override
23.      public void run(){
24.          for( int i=0; i<Conta.N_VALORES; i++ )
25.              Conta.incrementa();
26.              // Conta.aumenta();
27.      }
28.  }
29.
30.  class Consumer extends Thread {
31.      @Override
32.      public void run(){
33.          for( int i=0; i<Conta.N_VALORES; i++ )
34.              Conta.decrementa();
35.              // Conta.diminui();
36.      }
37.  }
```

2. COLLECTIONS

As principais estruturas de dados da computação são extremamente úteis para diversos contextos, contudo, sua implementação nem sempre é simples e rápida. Neste sentido, a linguagem de programação Java fornece nativamente o *Java Collections Framework*, que fornece componentes destas estruturas prontos e reutilizáveis, escritos para permitir a sua ampla reutilização.

Componentes: interfaces bem definidas e código bem estruturado.
API: uma biblioteca com muitas funções prontas para serem utilizadas.
Framework: fornece uma variedade de componentes que podem ser utilizados, mas também estendidas.

Uma coleção é uma estrutura de dados – um objeto – que pode armazenar outros objetos. As interfaces de coleção definem as operações que um programa pode executar sobre cada tipo de coleção. As principais interfaces são: Collection (coleções da maneira mais ampla possível), Set (conjunto), List (lista) e Map (mapeamento chave-valor). Cada interface é implementada por pelo menos uma classe, as quais foram cuidadosamente construídas a fim de permitirem rápida execução e uso eficiente da memória. O diagrama apresentado na Figura 2 mostra esta hierarquia.

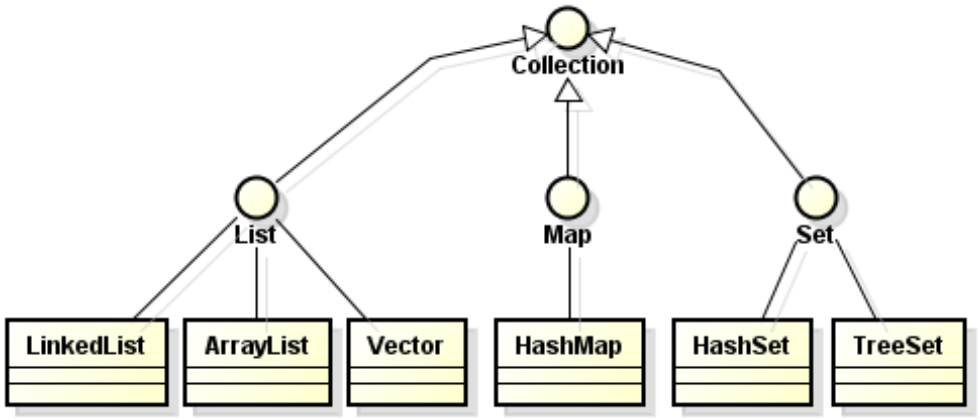

FIGURA 2 – Elementos do framework Java Collection

O Framework de coleções do Java utiliza o mecanismo de tipos genéricos (Generics) permitindo que coleções de tipos específicos ou genéricos sejam criadas pelo programador, não trazendo restrições e simplificando a reutilização de código.

2.1. Interface *Collection* e a Classe *Collections*

A interface *Collection* é a interface da raiz da hierarquia de coleções a partir da qual se derivam as demais interfaces mais restritivas de conjuntos, listas e mapas. A interface Collection é comumente usada como um tipo de parâmetro de métodos para permitir processamento polimórfico de todos os objetos que implementam a interface *Collection*.

A classe abstrata Collections, por sua vez, fornece algoritmos em forma de métodos estáticos para ordenar (sort), inverter (reverse), preencher (fill), copiar (copy), embaralhar conteúdo (shuffle), realizar buscas binárias (bynarySearch) e recuperar o maior e menor elemento (min, max), para citar alguns. Se tratando de uma classe abstrata, não é possível instanciar um objeto deste tipo, mas esta classe já implementa alguns métodos de forma que estes podem ser utilizados por classes derivadas.

2.2. Listas

A interface *List* representa uma coleção de elementos que não possui um tamanho máximo fixo de elementos (como um array) e que permite duplicação de elementos. Existem várias classes do framework de coleções do Java que implementam esta interface, mas destacamos quatro: ArrayList, LinkedList, ArrayDeque e Vector (obsoleto), sendo que cada uma possui implementação específica com características, atributos e métodos próprios.

A ArrayList é uma implementação da interface List como sendo uma lista redimensionável que não é sincronizada, ou seja, pode ser acessada por múltiplas threads de execução sendo mais rápidas, mas não possuem garantia de confiabilidade em inserções ou remoções. O Vector é outra implementação semelhante que também realiza o redimensionamento, mas é sincronizado, sendo mais lento do que um ArrayList, mas garantindo confiabilidade (atualmente obsoleto, pois garante a sincronização a cada operação de maneira atômica; dificilmente o que se deseja). A LinkedList, por sua vez, é uma implementação de lista, como uma lista encadeada, que também não é sincronizada, e o ArrayDeque é implementado como uma lista duplamente encadeada. Estes dois últimos possuem os métodos típicos das suas estruturas. O quadro abaixo ilustra um exemplo de uso.

Arquivo: Listas.java

```
 1.  package collections;
 2.  import java.util.ArrayDeque;
 3.  import java.util.ArrayList;
 4.  import java.util.Vector;
 5.  import java.util.LinkedList;
 6.
 7.  public class Listas {
 8.      public static void main(String[] args){
 9.          ArrayList<Integer> listaA = new ArrayList<>();
10.          Vector<String> listaB = new Vector<>();
11.          LinkedList listaC = new LinkedList();
12.          ArrayDeque listaD = new ArrayDeque();
13.
14.          listaA.add( 2 );
15.          listaB.size();
16.          listaC.clear();
17.          listaD.addLast( "teste" );
18.          Object obj = listaD.getLast();
19.      }
20.  }
```

2.3. Conjuntos

A interface *Set* representa uma coleção de elementos únicos (ou seja, não adiciona elementos duplicados) que não possui um tamanho máximo fixo de elementos. As duas principais classes que implementam esta interface são: HashSet e TreeSet, as quais possuem suas características de implementação específicas.

Hash: uma estrutura que utiliza uma função que converte um objeto qualquer em um índice. Assim, todo elemento é alocado em uma posição específica de um array (pode haver mais elementos em um mesmo índice), permitindo os benefícios de acesso imediato a posições do array.

A classe HashSet armazena os elementos em uma tabela de hash que não garante uma determinada ordem dos elementos, sendo que esta pode ser modificada durante as operações de inserção ou remoção de elementos. É uma estrutura que, na média, é muito eficiente para realizar operações de inserção, remoção e recuperação de elementos devido ao uso de hashing. Já a classe TreeSet implementa a interface *Set* como uma árvore, conforme o nome sugere, utilizando a comparação natural do tipo/classe especificada, ou utilizando uma classe que contem um método comparador personalizado e específico. As operações de inserção, remoção e recuperação de elementos é, em média, mais lenta do que no HashSet mas, a ordenação dos elementos é garantida – interessante quando se deseja mostrar a lista de elementos de forma ordenada.

Arquivo: Conjuntos.java

```
 1.  package collections;
 2.  import java.util.HashSet;
 3.  import java.util.TreeSet;
 4.
 5.  public class Conjuntos {
 6.      public static void main( String[] args ){
 7.          HashSet<String> conjuntoA = new HashSet();
 8.          TreeSet<Integer> conjuntoB = new TreeSet();
 9.
10.          conjuntoA.add( "teste" );
11.          conjuntoA.add( "poo" );
12.          boolean isThere = conjuntoA.contains( "teste" );
13.
14.          conjuntoB.add( 25 );
15.          conjuntoB.add( 3 );
16.          Integer lower = conjuntoB.first();
17.      }
18.  }
```

2.4. Mapeamentos

A interface *Map* representa uma estrutura de mapeamento baseada no conceito de chaves e valores, na qual todo elemento é representado por um par, e cada par possui um valor que se associa a sua chave. É uma estrutura semelhante à ideia de hash ou de um dicionário. Semelhante aos conjuntos, as duas principais classes que implementam esta interface são: HashMap e TreeMap.

A classe HashMap armazena os pares de elementos em uma tabela de hash com base no valor da chave (primeiro elemento do par), não garantindo uma determinada ordem dos elementos, sendo que esta pode ser modificada durante as operações de inserção ou remoção de elementos. É uma

estrutura que, na média, é muito eficiente para realizar operações de inserção, remoção e recuperação de elementos devido ao uso de hashing. Já a classe TreeMap implementa a interface *Map* como uma árvore rubro negra (*red-black*), utilizando a comparação natural do tipo/classe especificada, ou utilizando uma classe que contem um método comparador personalizado e específico. As operações de inserção, remoção e recuperação de elementos é, em média, mais lenta do que no HashMap mas, a ordenação dos elementos é garantida.

Arquivo: Mapeamento.java

```
 1.  package collections;
 2.  import java.util.HashMap;
 3.  import java.util.TreeMap;
 4.
 5.  public class Mapeamentos {
 6.      public static void main(String[] args){
 7.          TreeMap<Integer,Float> mapaA = new TreeMap();
 8.          HashMap<String,Integer> mapaB = new HashMap();
 9.
10.          mapaA.put( 25, 20.50f );
11.          float value = mapaA.get( 25 );
12.
13.          mapaB.put( "Carlos", 3 );
14.          mapaB.put( "Magnus", 1 );
15.          int position = mapaB.get( "Carlos" );
16.      }
17.  }
```

2.5. Pilhas e Filas

Outras duas estruturas fundamentais da computação são as filas e pilhas. Apesar delas poderem ser simuladas utilizando-se de listas, por exemplo, elas também estão disponíveis no pacote de utilitários do Java. Para fazer uso de uma estrutura de pilha, pode-se utilizar a classe Stack. A fila, por outro lado, é representada como uma interface, Queue, sendo efetivamente implementada por algumas classes nativas, entre elas a classe LinkedList, que já apareceu na subseção de listas, e a PriorityQueue (fila de prioridade), que além de funcionar como uma pilha, realiza a ordenação dos valores considerando a ordem natural ou uma classe comparadora. O quadro da próxima página apresenta um exemplo de uso destas estruturas.

2.6. Iteradores

Para diversas aplicações é usual ser necessário iterar por todos os registros armazenados em uma coleção. Mas como podemos fazer isso? Existem diversas formas, sendo que algumas delas são antigas e outras são relativamente recentes. As formas de iteração sobre elementos são:

- Uso do método *toArray()*: existem dois métodos toArray() para coleções que transferem os elementos armazenados na coleção para um array. Com os elementos transferidos, basta iterar sobre o novo array de elementos;
- Uso da interface *Iterator*: toda classe de coleção possui uma classe específica de iterador que implementa uma interface chamada *Iterator*. Isto é, toda coleção possui um método *iterator()* que devolve um objeto que permite a iteração sobre seus elementos, com base nos

métodos desta interface bem definida. Os principais métodos são: *hasNext()* que verifica se existe um próximo elemento na coleção e *next()* que retorna o próximo elemento. Note que se deve sempre utilizar o método *hasNext()* antes de *next()* para verificar a existência de um próximo elemento. Caso não exista, o método *next()* gera uma exceção;

- Uso da expressão *foreach*: as linguagens de programação modernas estão adicionando uma nova abordagem para o uso da estrutura de repetição *for*, que também é conhecida por *foreach*. Nas versões mais recentes do Java, a estrutura for pode ser utilizada com este intuito da seguinte forma: `for( tipo variável : coleção ){ ... }`, sendo que a cada iteração do laço de repetição, a variável do laço receberá um novo valor da repetição;
- Uso de expressão funcional: o aumento do interesse por linguagens funcionais fez com que muitas linguagens de outros paradigmas adicionassem conceitos funcionais. Entre um dos conceitos funcionais introduzidos em Java, estão as funções anônimas. Para fazer uso da iteração funcional, as coleções fornecem um método chamado *forEach()* que recebe como parâmetro um objeto da classe Consumer. Um objeto desta classe pode ser escrito como uma função anônima. É uma forma de se passar uma função como parâmetro para um método.

Um exemplo ilustrativo que indica a sintaxe destes métodos é apresentado no quadro que apresenta a classe Iteradores.java.

Arquivo: Estruturas.java

```
 1.  package collections;
 2.  import java.util.PriorityQueue;
 3.  import java.util.Queue;
 4.  import java.util.Stack;
 5.
 6.  public class Estruturas {
 7.      public static void main(String[] args){
 8.          Stack<Integer> pilha = new Stack();
 9.          Queue<Double> fila = new PriorityQueue();
10.
11.          pilha.push( 5 );
12.          pilha.push( 2 );
13.          int top = pilha.pop();
14.
15.          fila.add( 9.3 );
16.          fila.add( 7.26 );
17.          double front = fila.peek();
18.      }
19.  }
```

Arquivo: Iterador.java

A fim de reduzir o espaço ocupado pelas inserções foram colocados dois comandos por linha. **Não faça** isso em casa ou na faculdade ou no trabalho!

```
 1.  package collections;
 2.  import java.util.Iterator;
 3.  import java.util.TreeSet;
 4.
 5.  public class Iterador {
 6.      public static void main(String[] args){
 7.          TreeSet<String> tree = new TreeSet();
 8.          tree.add( "Marli" );    tree.add( "Zeca" );
 9.          tree.add( "Alison" );   tree.add( "Pablo" );
10.          tree.add( "Josias" );   tree.add( "Fernanda" );
11.          tree.add( "Zeca" );     tree.add( "Beatriz" );
12.
13.          System.out.println("\n--- ARRAY ---");
14.          Object[] array = tree.toArray();
15.          for(int i=0; i<array.length; i++)
16.              System.out.println( (String) array[i] );
17.
18.          System.out.println("\n--- ITERATOR ---");
19.          Iterator<String> it = tree.iterator();
20.          while( it.hasNext() ){
21.              String nome = it.next();
22.              System.out.println(nome);
23.          }
24.
25.          System.out.println("\n--- FOREACH ---");
26.          for(String nome : tree)
27.              System.out.println(nome);
28.
29.          System.out.println("\n--- FUNCIONAL ---");
30.          tree.forEach( (nome) -> {
31.              System.out.println(nome);
32.          });
33.      }
34.  }
```

Exercícios

1) Elabore um programa que gere uma matriz A com dimensão 1000 x 1000 com números inteiros aleatórios entre 0 e 9. Implemente então um método *monothread* de multiplicação entre matrizes e utilize-o para calcular A*A. Agora crie uma nova classe para calcular a mesma multiplicação utilizando 4 threads e verifique a diferença no tempo de execução.

DICA: Note que é possível separar a tarefa de multiplicação entre os threads de diversas formas. É possível determinar que cada thread fique responsável por multiplicar uma parcela sequencial de linhas por colunas, por exemplo, o thread 1 multiplica da linha 0 até a linha 250, o thread 2 multiplica da linha 251 até a linha 500 e assim sucessivamente. Ou então que cada thread multiplique linhas intercaladas, por exemplo, o thread 1 multiplica todas as linhas que ao dividir o seu número por 4 fique resto 1; o thread 2 multiplica todas as linhas que ao dividir o seu número por 4 fique resto 2; e assim sucessivamente. Lembre-se de especificar estes valores em um construtor ou um método que seja invocado antes da execução do thread.

2) Implemente um sistema de cadastro de mercadorias e cálculo do preço de compras. O sistema deverá ter duas três opções:

(1) Cadastrar item: o sistema deverá perguntar o nome, descrição e valor de um produto e armazenar o item em uma coleção.

(2) Excluir item: o sistema deverá perguntar qual o nome do item a ser descartado. Se um produto com o nome especificado existir ele deve ser excluído da coleção, caso contrário deve-se mostrar uma mensagem de erro.

(3) Compra: quando uma compra for iniciada o sistema deverá iniciar um registro de compra. O sistema deve perguntar repetidas vezes qual o nome do item que está sendo comprado e a quantidade. Ao digitar um nome correto o item é adicionado no "carrinho de compra virtual". Para finalizar uma compra o sistema deverá requisitar a palavra-chave FIM. Caso um item tenha sido cadastrado no carrinho por engano deve-se utilizar a palavra-chave APAGAR para remover o último item da compra. Ao final da compra, o sistema deve mostrar uma prévia da notinha fiscal, mostrando item a item, o nome, o peso e a quantidade. Ao final da nota, deve-se mostrar o peso total dos itens e o valor total da compra (considerando as quantidades de cada item).

Procure escolher as melhores estruturas para realizar as operações mencionadas.