PROGRAMAÇÃO ORIENTADA A OBJETOS

AULA 3

Prof. Leonardo Gomes

CONVERSA INICIAL

Nesta aula, vamos abordar a questão da visibilidade nas classes. Nas linguagens de programação estruturadas, existem diferentes técnicas para limitar quem tem acesso a determinadas funções e variáveis. Essa estratégia permite, entre outras coisas, esconder atributos e métodos que são utilizadas apenas internamente. Na orientação a objetos, isso é feito por meio de modificadores próprios que veremos em detalhes na sequência.

Também vamos discutir algumas bibliotecas importantes dentro do Java que podem facilitar o desenvolvimento de aplicações.

Objetivos da aula: ao final desta aula, esperamos atingir os seguintes objetivos que serão avaliados ao longo da disciplina da forma indicada.

Quadro 1 – Objetivos

Objetivos	Avaliação
1 – Aplicar os conceitos de encapsulamento dentro da orientação a objetos.	Questionário e questões dissertativas
2 – Desenvolver algoritmos que fazem uso correto dos modificadores *public* e *private*.	Questionário e questões dissertativas
3 – Conhecer outras classes internas e importantes ao Java.	Questionário e questões dissertativas

TEMA 1 – VISIBILIDADE

Neste tema, vamos debater o conceito de visibilidade por meio dos modificadores ***public***, ***protected*** e ***private*** que definem quais outras classes podem acessar seus atributos e métodos.

Na orientação a objetos, à medida que os projetos crescem em tamanho, desenvolvemos um grande número de classes e se torna cada vez mais difícil lembrar de todos os métodos e atributos

pertinentes para cada situação.

Neste sentido, como já discutimos em aulas anteriores, é importante aplicarmos boas práticas na programação como escolher bons nomes para todos os elementos da classe e seguir padronizações no estilo de codificação.

Outra prática que nos ajuda a simplificar o uso das classes é esconder métodos e atributos que sejam de uso interno da classe, de forma que, ao fazermos uso da classe posteriormente, não seja exibido detalhes de implementação, e sim apenas os métodos e atributos que sejam pertinentes.

Antes da definição de cada método, atributo e classe, podemos colocar um dos três modificadores:

- ***Public***: o elemento é público e pode ser acessado por qualquer outra classe sem restrições.
- ***Private***: o elemento é privado e só pode ser acessado internamente na classe.
- ***Protected***: o elemento é protegido e será acessado somente de dentro da própria classe, outras classes no mesmo pacote e também por classes filhas. A definição de classe filha será abordada em detalhes na aula de herança.
- ***Default* (sem nenhum modificador)**: o elemento, neste caso, é acessível por classes dentro do mesmo pacote.

Tabela 1 – Tabela demonstrativa dos modificadores de visibilidade

Visibilidade	*Public*	*Protected*	*Default*	*Private*
Mesma classe	SIM	SIM	SIM	SIM
Classe qualquer no mesmo pacote	SIM	SIM	SIM	NÃO
Classe filha e mesmo pacote	SIM	SIM	SIM	NÃO
Classe filha e pacotes diferentes	SIM	SIM	NÃO	NÃO
Classe qualquer em outro pacote	SIM	NÃO	NÃO	NÃO

1.1 EXEMPLO COM CÓDIGO

No exemplo a seguir, temos um código que demonstra o uso dessas palavras reservadas.

```
01.     public class Aluno {
02.          private int matricula;
03.          protected int notas[];
04.          public String cpf;
05.          public String nome;
06.
07.          public Aluno(String nome, String cpf){
08.             this.nome = nome;
09.             this.cpf = cpf;
10.           this.matricula =
   Cadastro.gerarNovaMatricula();
11.          }
12.          public void cadastrarNotas(){
13.             //codigo cadastro
14.          }
15.          public int mediaNotas(){
16.           //codigo media
17.          }
18.
19.     }
```

Em relação ao código acima, digamos que o atributo *matrícula* não seja uma informação de interesse para quem utiliza a classe *Aluno*, mas que internamente na classe é uma informação importante para talvez vincular o aluno em um banco de dados, por exemplo. Para essas situações, atributos como *matricula* podem ser definidos como privados, garantindo uma abstração que invisibiliza informações que sejam pertinentes apenas para o contexto interno da classe.

Ainda no exemplo, os atributos *cpf* e *nome,* por serem pertinentes para quem for utilizar a classe, ficam públicos. Observe que o atributo *notas* foi definido como *protected*, por algum motivo poderia não ser interessante permitir seu público e direto ao *array* que guarda os dados, mas ao mesmo tempo suponha que existam algumas classes específicas que gostaria de oferecer acesso, nesta situação, nem *public* e nem *private* atendem a necessidade, *protected* concede visibilidade apenas

para classes relacionadas a classe *Aluno*, mais especificamente suas classes filhas, o conceito de classe filha e *Herança* será visto em detalhes nas próximas aulas.

1.2 VISIBILIDADE NO DIAGRAMA DE CLASSES UML

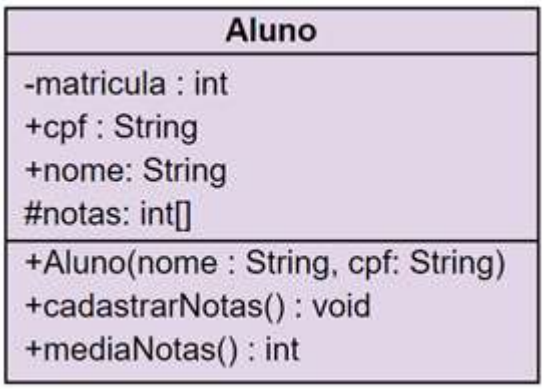
Nas aulas anteriores, demonstramos como representar classes utilizando diagrama de classes UML. Nesse diagrama, podemos também representar a visibilidade dos métodos e atributos utilizando os seguintes símbolos:

Quadro 2 – Símbolos e significados

Símbolo	Significado
+	Público
-	Privado
#	Protegido

A seguir, o exemplo da classe *Aluno*.

Figura 1 – Classe *Aluno*



Desafio: seguindo o exemplo da classe *Aluno*, desenvolva o diagrama de classe UML para descrever um ITEM qualquer que seja comercializado por uma loja on-line, não existe uma resposta certa, cada contexto requer atributos e métodos diferentes, mas pense como imagina sendo a melhor forma.

Quais informações você acredita serem importantes armazenar para exibir ao comprador e registrar a venda na loja? Quais atributos imagina que seriam privados por serem utilizados apenas internamente? E quais seriam públicos por interagir com outros objetos?

TEMA 2 – ENCAPSULAMENTO

Entende-se que o Paradigma Orientado a Objetos possui três pilares, **herança**, **polimorfismo** e o **encapsulamento**. Alguns autores defendem a **abstração** como um quarto pilar, no entanto, ela pode também ser entendida como parte do encapsulamento. Neste tema, vamos debater em mais detalhes os benefícios e práticas do encapsulamento.

Quando pensamos em uma cápsula no mundo real, fora do contexto da programação, imaginamos o invólucro de algum objeto sensível e/ou perigoso, que serve tanto para proteger o conteúdo interno do mundo exterior quanto o mundo exterior desse conteúdo interno.

Se pensarmos no monitor para computadores, ele é um exemplo de encapsulamento. Seus componentes ficam resguardados pela estrutura de plástico que o envolve, evitando que quem manipula o equipamento receba descargas elétricas ou danifique o aparelho.

Na programação orientada a objetos, tentamos abstrair no código a interação entre os objetos reais, portanto, a lógica do encapsulamento é a mesma, como boa prática de programação orientada a objetos, devemos utilizar as propriedades *public*, *private* e *protected* para invisibilizar os componentes internos das classes que não são pertinentes e deixar visível o estritamente necessário. Dessa forma, o programador que fará uso da classe encapsulada tem uma camada extra de segurança no momento que fizer uso da classe.

No contexto da computação, chamamos de interface os elementos responsáveis por conectar de forma física ou lógica diferentes objetos ou partes distintas de um sistema. No caso do exemplo dado do monitor, a conexão hdmi compõe sua interface, já no caso de uma classe sua interface se dá por meio dos seus métodos que recebem e devolvem informações.

Em uma classe que segue boas práticas de encapsulamento, seria possível trocar a classe por outra que possua a mesma interface (métodos com os mesmos nomes e parâmetros) e, por mais que as suas implementações internas mudem, o sistema continuará funcional sem maiores mudanças.

No mundo físico, mais uma vez o exemplo do monitor, se trocarmos o equipamento por outro, o sistema continua funcionando sem maiores complicações, desde que ambos possuam a mesma interface (cabo HDMI) basta desconectar um e conectar o outro, mesmo que sejam monitores com especificações diferentes (tecnologia LCD ou LED, tamanhos diferentes etc.).

Já no contexto da computação acontece o mesmo se temos duas classes com uma mesma interface, mas que internamente resolve códigos de maneira distintas, por exemplo, utilizando diferentes estruturas de dados ou aplicando diferentes algoritmos para ordenarem seus dados, o restante do código ficaria inalterado. O que possibilita inclusive diferentes equipes que não mantém contato algo produzirem soluções que conversem. Tal qual o exemplo do monitor que mesmo quando produzidos por uma empresa diferente do restante do equipamento continua funcionando.

Em resumo, a vantagem do encapsulamento são:

1. A abstração oferecida em que o funcionamento interno dos objetos da classe não fica visível ao programador que utiliza a classe.
2. A possibilidade de acrescentar funcionalidades à classe desde que respeitando a interface original manterá o sistema funcional sem alterações.
3. Simplificação da utilização dos objetos em um alto nível acelera o desenvolvimento dos códigos.
4. O sistema fica robusto a mudanças internas, mesmo uma substituição completa do código que poderia até mesmo ser desenvolvido por outra equipe que não manteve contato com a primeira, bastando respeitar a interface.

2.2 EXEMPLO JAVA DO PARADIGMA ORIENTADO A OBJETO

Uma das práticas do encapsulamento consiste em criar métodos chamados de *getter* e *setter* para conceder acesso aos atributos da classe. A ideia está em deixar todos os atributos (variáveis internas da classe), como privadas e aquelas que convém ao usuário acessar são disponibilizadas por meio desses métodos.

No inglês, *get* significa pegar, e os métodos *getters* são métodos que pegam os valores daqueles atributos e retornam para quem chamou o método enquanto *set* significa definir, servindo justamente para definir os atributos com o valor passado por parâmetro. Essa lógica é aplicada às mais diversas linguagens de programação com pequenas variações, mas a ideia de restringir ao

acesso, encapsulando a classe, é a mesma. No caso do Java, vejamos um exemplo a seguir com uma classe que representa horários.

```
01.     public class Horario {
02.       private int hora;
03.       private int minuto;
04.       private int segundo;
05.
06.      public void setHora(int h) {
07.        if (h > 23 || d < 0) {
08.          System.out.println("Valor inválido");
09.        }else {
10.          hora = h;
11.        }
12.      }
13.
14.            public int getHora() {
15.                return hora;
16.            }
17.     }
```

Observe o padrão de nomenclatura das funções *setHora* e *getHora* para o atributo hora, esse padrão é amplamente adotado na comunidade de programadores sempre com a mesma funcionalidade, definir e retornar o valor para determinado atributo.

No caso, é possível inclusive realizar validação no momento de definir o dado para garantir a integridade a informação. Neste exemplo, não seria possível definir uma hora fora do intervalo 0-23. Suponha que essa classe seja adota em aplicações críticas como as bancárias então garantir a integridade se torna ainda mais importante.

Desafio: desenvolva uma classe Data, com atributos, dia, mês e ano encapsulados com *get/set*. Para simplificar, assuma que todo mês pode seja composto por 30 dias.

TEMA 3 – COLLECTIONS

Neste Tema, vamos debater uma importante API em Java chamada *Collections*, essa API consiste em um conjunto de classes que implementam diferentes estruturas de dados, essas estruturas são encapsuladas respeitando uma mesmo acordo, interface, o que traz diversos facilidades.

Por *estrutura de dados* entenda a estratégia que você utiliza para organizar seus dados na memória. Você certamente deve estar familiarizado com a estrutura de dados *array* (ou lista) e matriz.

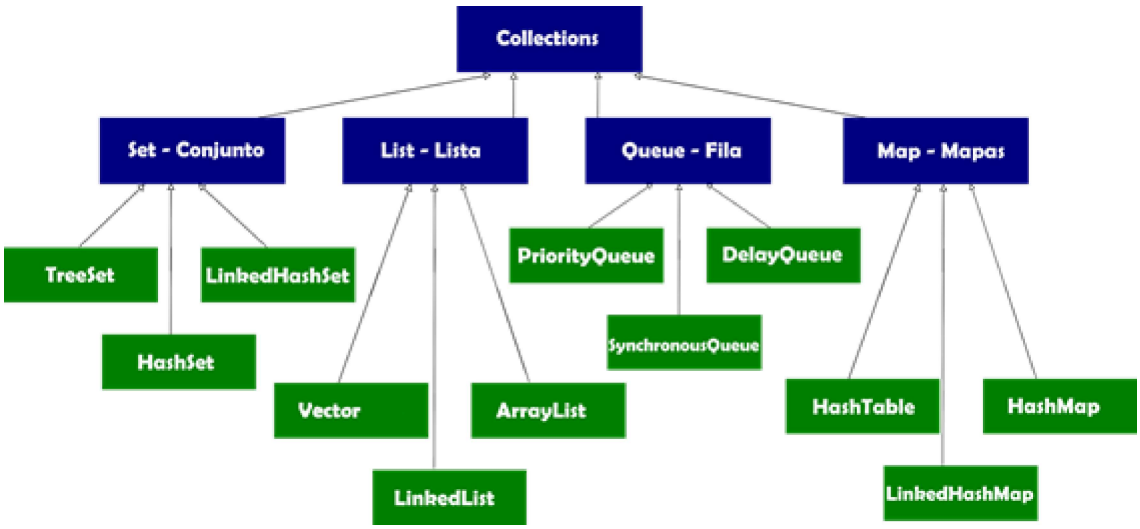
Porém, existem diversas outras estruturas, como pilha, fila, *hash*, árvores, lista encadeada, entre outros. O estudo de estrutura de dados é todo um campo dentro da ciência da computação com diversas ramificações e que tem uma importância muito grande, adotar a estrutura de dados correta em cada situação pode ser a diferença entre um algoritmo que leva dias para executar ou poucos minutos.

No entanto, implementar as estruturas de dados adequadamente toma tempo e, por possuir soluções bem conhecidas, a API *Collections* já nos traz as classes prontas com essas estruturas; dessa forma, não precisamos “reinventar a roda”.

3.1 PRINCIPAIS CLASSES DA *COLLECTIONS*

Dentro das *Collections,* existem diversas estruturas de dados, de forma geral, elas são divididas em quatro grupos, dispostos em azul na Figura 1. O grupo *list* (listas), que funciona como uma sequência ordenada de valores, o grupo *set* (conjuntos), que forma um agrupamento bde itens sem ordem definida, o grupo *map* (mapas), que mapeia dados chamados chaves para outros dados chamados de valores e, por fim, o grupo *queue (*filas), que implementam um tipo de *array* em que a posição de cada elemento define a prioridade dele em relação aos demais, uma abstração chamada de fila de prioridades. Na Figura 2, vemos os quatro grupos e suas classes e, ao longo do texto, descrevemos algumas das que consideramos mais importantes.

Figura 2 – Tabela demonstrativa dos modificadores de visibilidade



ArrayList é uma das classes mais comuns dentro da API *Collections*, como já apresentada anteriormente ele representada um *array* dinâmico. Os elementos dentro dela possuem uma ordem definida e trazem métodos de manipulação, como remoção, inserção, busca, entre outros.

LinkedList é outra classe que também implementa um *array* dinâmico com as mesmas funcionalidades do *ArrayList* tradicional, no entanto, sua implementação interna utiliza a estratégia de lista ligada e realiza as operações de remoção e inserção de forma muito mais rápida e a busca por elementos de forma muito mais lenta em comparação ao *ArrayList* tradicional.

De maneira geral, é mais comum ao longo da execução de um programa que mais buscas sejam executadas do remoções/inserções, portanto, o *ArrayList* de forma geral é o mais indicado, entretanto, para aqueles casos em que isso não ocorre, contamos com essa alternativa.

HashSet na computação, o termo *Hash* diz respeito a uma função que transforma um valor em outro, e a palavra *Set* tem diversos significados diferentes sendo que, nessa situação específica, significa conjunto. Então uma *HashSet* é um conjunto de elementos organizados por meio de uma função *Hash*. Ela realiza operações de adição, remoção e busca de forma muito rápida, contudo, não garante uma ordem dos elementos.

LinkedHashSet é semelhante a *HashSet*, porém, ele armazena a ordem em que os elementos foram adicionados.

HashMap é a estrutura de dados também baseada em *Hash*, com a diferença que é possível mapear (daí o nome *Map*) uma ID de um tipo diverso. Por exemplo, podemos ordenar os filmes de

um catálogo não por um valor inteiro, mas por uma *string* contendo o nome do filme seguido do ano de exibição. Não possui ordem garantida.

TreeMap semelhante a *HashMap*, entretanto, também armazena a ordem dos itens, essa ordem pode ser livremente manipulada combinando as características de em um *array*.

LinkedHashMap é semelhante a *HashMap,* todavia, internamente também armazena a ordem em que os elementos foram adicionados.

Queue é uma estrutura de dados geralmente adotada para representar filas de prioridade, *queue* no inglês significa fila. Ela pode implementar uma fila de prioridade comum, semelhante a uma fila convencional no mundo real em que o primeiro elemento a entrar é o primeiro a ser atendido.

Stack é outra classe semelhante a fila de prioridades, mas implementa uma fila reversa à ordem de inserção, o último elemento adicionado é o primeiro a ser tratado, conceito denominado pilha, se empilharmos diversos objetos um sobre o outro, o último objeto empilhado será o primeiro que vamos acessar.

3.2 EXEMPLO DE USO DE *ARRAYLIST/LINKEDLIST*

A seguir, demonstramos alguns métodos aplicados nas classes *ArrayList* e *LinkedList*.

```
    //Declaração do array, tanto arraylist quanto linked list
possuem métodos com a mesma assinatura permitindo um mesmo
código funcionar da mesma forma modificando apenas a estrutura
de dados escolhida.
    ArrayList<String> pessoas = new ArrayList<String>();
    //LinkedList<String> pessoas = new LinkedList<String>();

    pessoas.add("Mario"); //Adição de novos elementos
    pessoas.add("Luigi");
    pessoas.add("Peach");
    pessoas.add("Yoshi");
    System.out.println(pessoas);//Lista dos elementos
    item1 = pessoas.get(0); //retorna o elemento de índice 0
    pessoas.remove(3);//remove o elemento de índice 3
    total = pessoas.size();//retorna a quantidade de elementos
    pessoas.clear(); //Remove todos os elementos
```

3.3 EXEMPLO DE USO DE *HASHMAP*

A seguir, o código demonstra o uso da classe *HashMap*. As classes baseadas na estrutura de mapa possuem o conceito de chave e valor. A chave é o que indexa ela e o valor o conteúdo dentro daquele índice.

No código a seguir, o *HashMap* capitais indexa capitais (valor) por meio do nome do país (chave). É diferente de termos um *array* composto de uma *struct* contendo o par de *strings*, pois, nesse caso, o índice continuaria sendo um inteiro, marcando a posição, já na *Hash,* o que marca a posição é o próprio nome da capital.

```
HashMap<String, String> capitais = new HashMap<String,
String>();
capitais.put("Brasil", "Brasília");
capitais.put("Argentina", "Buenos Aires");
capitais.put("Paraguai", "Assunção");
capitais.put("Uruguai", "Montevidéu");
System.out.println(capitais);//Imprimindo tudo

System.out.println(capitais.get("Uruguai"));//Imprimindo  apenas
a capital do Uruguai
```

3.4 EXEMPLO DE USO DE *HASHSET*

A seguir, o código demonstra o uso da classe *HashSet*. As classes baseadas na estrutura de Conjunto armazenam dados sem se preocupar em manter uma ordem, o conceito de ordem em conjunto simplesmente não existe, se adicionarmos um item primeiro, ele não estará na frente do que adicionarmos por segundo. Sem ter a necessidade de se preocupar com ordem, certos métodos se tornam mais rápidos, por exemplo, descobrir se determinado elemento existe ou não é muito mais rápido em uma estrutura de dados de Conjunto em comparação à Lista.

```
HashSet<String> nomes = new HashSet<String>();
nomes.add("Mario");
nomes.add("Luigi");
nomes.add("Yoshi");
nomes.add("Mario");//Mario já existe portanto não será
adicionado
nomes.add("Peach");
nomes.remove("Luigi");//remove luigi
System.out.println(nomes);//Imprime todos os nomes
int total = nomes.size();//descobre total de itens
if(nomes.contains("Luigi")) {//Confere se existe
   System.out.println("Ele está presente");
}
else {
   System.out.println("Não está presente");
}
```

3.5 EXEMPLO DE USO DE *PRIORITYQUEUE*

A seguir, o código demonstra o uso da classe *PriorityQueue*. As classes baseadas no grupo das filas de prioridade armazenam dados se preocupando de forma muito restrita com a ordem, existem filas em que desejamos visualizar os elementos conforme a ordem que foram adicionados (fila tradicional) ou na ordem reversa ao que foram adicionados (pilha).

É indicado adotar esse tipo de estrutura quando temos dois tipos de entidades distintas, um tipo produzindo algum tipo de dado e outro consumindo os dados. O exemplo clássico é a gestão de documentos de uma impressora. Podemos ter vários serviços gerando dados para serem impressos e existe a impressora que consome (imprime) os dados na ordem que foram adicionados na fila.

```
// Criando a fila
PriorityQueue<Integer> fila = new PriorityQueue<Integer>();

// adicionando elementos para fila usando add()
fila.add(10);
fila.add(20);
fila.add(15);
// Imprimindo o elemento do topo da fila
System.out.println(fila.peek());

//Imprimindo e ao mesmo tempo removendo o primeiro elemento
System.out.println(fila.poll());

// Imprimindo o elemento do topo novamente
System.out.println(fila.peek());
```

3.6 MÉTODOS ESTÁTICOS *COLLECTIONS*

Além das classes, existem vários métodos estáticos dentro do *framework Collections* que implementam soluções para problemas que são relativamente comuns entre os algoritmos. Por exemplo, o método *sort* implementa a ordenação dos elementos que compõem um objeto da classe *Collections*, ou seja, podemos com uma única chamada colocar os itens de um *ArrayList* em ordem crescente. A seguir, uma lista com exemplos de alguns dos métodos mais importantes:

- *Sort* (*List*<> lista): coloca em ordem crescente os itens da lista passada por parâmetro.
- *Shuffle* (*List*<> lista, *Random* rnd): embaralha de forma aleatória os elementos da lista passada por parâmetro, a aleatoriedade do embaralhamento é dada pelo objeto da classe *Random* passado por parâmetro também.
- *Max* (*Collection*<> *coll*, *Comparator*<> comp): retorna o maior elemento, aceita tanto lista, quanto *hash*. Como segundo parâmetro, você pode indicar como deseja realizar a comparação com um objeto da classe *Comparator*, caso passe *null* como segundo parâmetro, a ordem natural será adotada.

- *Min* (*Collection*<> *coll*, *Comparator*<> comp): análogo ao *max*, porém, retorna o menor elemento.
- *Reverse* (*List*<> lista): coloca todos os itens em ordem reversa.

O código a seguir exemplifica o uso desses métodos:

```
// Cria uma lista de string
ArrayList<String> lista = new ArrayList<String>();
lista.add("Mario");
lista.add("Luigi");
lista.add("Yoshi");
lista.add("Toad");
lista.add("Peach");

//Imprime a lista na ordem original
System.out.println(lista);

//Coloca lista em uma ordem aleatória
Collections.shuffle(lista);
System.out.println(lista);

//Coloca lista em uma ordem alfabética
Collections.sort(lista);
System.out.println(lista);

//Inverte a ordem anterior da lista
Collections.reverse(lista);
System.out.println(lista);

//Maior elemento alfabético
System.out.println("Maior: "+ Collections.max(lista));

//Menor elemento alfabético
System.out.println("Menor: "+Collections.min(lista));
```

TEMA 4 – *ITERATOR*

Neste tema, vamos discutir formas de navegar pelos dados de uma das classes presentes no *framework Collections*, vamos dar destaque especial a uma estratégia chamada *Iterator*.

Quando desejamos visitar os dados em uma estrutura de dados, as estratégias mudam dependendo da estrutura, listas contam com índices inteiros, mapas são indexados pelas chaves que foram definidas, conjuntos não possuem forma de indexação alguma. No entanto, os *iterators* são uma ferramenta poderosa nesse sentido, pois com eles é possível navegar pelos dados independentes da classe *Collections* utilizada. Vamos ver alguns exemplos de código.

```
01.     ArrayList<Integer> lista = new ArrayList();
02.     HashSet<Integer> conjunto = new HashSet<Integer>();
03.     HashMap<String,Integer> mapa = new
04.     HashMap<String,Integer>();
05.     int soma;
06.
07.     soma=0;//For simples
08.     for(int i=0;i<lista.size();i++) {
09.          soma += lista.get(i);
10.     }
11.
12.     soma=0;//For each
13.     for(int item : lista) {
14.          soma += item;
15.     }
16.
17.     soma=0;//Iterator
18.     //Iterator it = mapa.entrySet().iterator();
19.     //Iterator it = conjunto.iterator();
20.     Iterator it = lista.iterator();
21.     while(it.hasNext()) {
22.          soma += (int)it.next();
23.     }
```

Nesse código, temos três formas diferentes de calcular a soma dos elementos de um *ArrayList*, vamos discutir cada uma dessas formas:

1 - Das linhas de código 7 até 10, temos a forma tradicional, utilizando um *for* indexando a lista pelos seus índices indo de 0 até o tamanho total da *lista.size()*.

2 - Na sequência, temos o que chamamos de um *for each,* nas linhas 12 até 15, um comando de repetição *for* que em vez de ser dividido em três partes é dividido em duas e que dispensa o uso de índices, ele executa o *loop* para cada elemento daquele *Collection* que for passada depois dos dois pontos, no caso lista. A cada execução, a variável que vai antes dos dois pontos, no caso item, receberá um valor distinto. Esse tipo de *for* é interessante por ser simples e também se aplicar a diferentes classes dentro das *Collections*, no entanto, ela é bastante engessada não permitindo operações mais elaboradas, como remover um item específico que foi visitado, entre outras operações.

3 - Na sequência, linha 17 em diante, temos a demonstração do uso do Iterator. Ele é compatível com diversos estruturadas de dados diferentes, nas linhas 18, 19 e 20, vemos o iterator sendo utilizado respectivamente com mapa, conjunto e lista. O código é o mesmo para todos. o iterator é uma espécie de ponteiro para o elemento da *Collection*. Na linha 21, o método *hasNext()* verifica se existe um próximo elemento e retorna verdadeiro ou falso. Na linha 22, o método *next()* atualiza o iterador para o próximo item da coleção. Como os ponteiros iteradores podem apontar para qualquer tipo primitivo ou objeto, é necessário indicar ao compilador como aquele item deve ser interpretado, no caso *(int)* é o que chamamos de *cast-type* na programação e serve para indicar ao compilador que o comando que vem na sequência deve ser lido como um inteiro. Importante observar que o *cast-type* não é uma conversão, ele não modifica o dado, apenas muda a forma como aquele dado é visualizado pelo compilador, o uso inadequado desse comando pode gerar exceções na execução do programa.

O *iterator* é bastante simples e compatível com diversas estruturas de dados, o que o torna particularmente útil para criar um código flexível e independente da estrutura de dados adotada. Além do exemplo visto no código anterior, seria possível também navegar de forma reversa, de trás para frente como no código a seguir.

```
//Suponha lista como um arrayList com dados de algum
tipo, por exemplo strings.
1. ListIterator it = lista.listIterator(lista.size());
2. while (it.hasPrevious()) {
3.     System.out.println(it.previous());
4. }
```

Observe que aqui é utilizada a classe *ListIterator* em vez de apenas *Iterator,* como visto no exemplo anterior. Neste caso, é necessário especificar que o *iterator* age sobre listas, pois nelas existe a ideia de uma ordem sequencial dos dados, em outras estruturas, como *Map* e *Set,* essa ordem não existe por padrão.

O método *listIterator* (linha 1) aceita como parâmetro uma posição, 0 para primeira posição, 1 para segunda posição e assim por diante. No caso, o tamanho da lista é o parâmetro, portanto, o retorno será um iterator que aponta para a última posição.

O método *hasPrevious()* do inglês se traduz como "existe um anterior?" e como o nome sugere ele retorna *true* ou *false* como resposta. Por sua vez, o método *previous()*, do inglês "anterior", retorna o *iterator* em si da posição anterior. Em outras palavras, enquanto existir um elemento anterior, o programa segue imprimindo na tela o anterior do anterior do anterior e assim por diante.

Também é possível remover elementos apontados pelo *Iterator*, segue novo código abaixo.

```
01.     import java.util.ArrayList;
02.     import java.util.Iterator;
03.     public class Exemplo {
04.        public static void main(String[] args) {
05.           ArrayList<String>lista= new
   ArrayList<String>();
06.           lista.add("Maca");
07.           lista.add("Manga");
08.           lista.add("Abacate");
09.           lista.add("Laranja");
10.           lista.add("Pessego");
11.           System.out.println("Os elementos: ");
12.           for (String s: lista) {
13.              System.out.println(s);
14.           }
15.           Iterator i = aList.iterator();
16.           String str = "";
17.           while (i.hasNext()) {
18.             str = (String) i.next();
19.             if (str.equals("Laranja")) {
20.               i.remove();
21.               System.out.println("Laranja removida");
22.               break;
23.             }
24.           }
25.           System.out.println("Os elementos: ");
26.           for (String s: lista) {
27.             System.out.println(s);
28.           }
29.        }
30.     }
```

No código acima, temos uma lista de *strings* com nomes de frutas e todos os elementos são impressos na tela, antes e depois da *string* "Laranja" ser removida da lista. Na linha 17, vemos o *loop* utilizando *while* e o método "*hasNext()*" já discutido anteriormente, na linha 19, fazemos uma comparação para decidir se a *string* que o *iterator* aponta é realmente "Laranja", se for o caso, ela é removida com o método *remove()*, então o *loop* é encerrado utilizando a instrução *break*.

TEMA 5 – CLASSE *LOCALDATE*

Em diversos projetos, nos deparamos com o desafio de como lidar com a representação de datas e horários, essa é uma questão especialmente comum em projetos quando envolve banco de dados e acesso *web*. O Java conta com algumas soluções implementadas internamente. Neste Tema, vamos discutir as principais soluções.

Originalmente, nas primeiras versões do Java, foi implementada uma classe para datas chamada *java.util.Date,* essa classe originalmente possuía diversas limitações e, por isso, em uma atualização do Java, surgiu a classe *java.util.Calendar,* que também não atendia às necessidades da comunidade de desenvolvimento Java, por isso, se popularizou muito uma biblioteca chamada *Joda Time*.

A Oracle, ao identificar a preferência da comunidade pela biblioteca externa na versão 8, integrou esta ao Java com o nome *java.util.LocalDate* (e suas derivações). Portanto, a recomendação é a utilização da *LocalDate* para versão 8 do Java em diante e, se por alguma restrição específica de projeto for necessário utilizar uma versão antiga do Java, recomenda-se o uso da Joda Time. A seguir, o código que representa o uso do *LocalData*

```
01.     public static void main(String[] args) {
02.
03.         //Captura a data de hoje
04.         LocalDate dataHoje = LocalDate.now();
05.
06.         System.out.println("Original: " + dataHoje);
07.         DateTimeFormatter formatador =
   DateTimeFormatter.ofPattern("dd/MM/yyyy");
08.         String dataForm= hoje.format(formatador);
09.         System.out.println("Formatado : " + dataForm);
10.
11.     }
```

A classe *LocalDate* representa datas de forma bem prática. Cada data sendo um objeto *LocalDate*, para, por exemplo, recuperarmos a data do momento da execução de uma linha de código. *LocalDate.now*();(linha 4), que significa "agora" em inglês.

Se imprimirmos na tela objeto do tipo *LocalDate,* teremos a impressão no formato americano que coloca o mês na frente do dia. Para representar no formato brasileiro, é necessário criar um objeto do tipo *DateTimeFormatter,* que estabelece formatações para datas e horários. Na linha 7 do código, vemos a formatação estabelecida sendo dd/MM/yyyy, que representa a data na ordem que estamos acostumados e separadas por uma barra.

- dd = dia do mês em dois dígitos;
- MM = mês em dois dígitos;
- yyyy = ano em quatro dígitos;
- HH = horas, até 23, em dois dígitos;
- mm = minutos em dois dígitos;
- ss = segundos em dois dígitos;
- hh = horas, até 12, em dois dígitos;
- d = dia do mês em um ou dois dígitos;
- M = mês do ano em um ou dois dígitos;

- yy = ano em dois dígitos;
- H = horas, até 23, em um ou dois dígitos;
- m = minutos em um ou dois dígitos;
- s = segundos em um ou dois dígitos;
- h = horas, até 12, em um ou dois dígitos.

Quando desejamos trabalhar com horário, o procedimento é semelhante. Confira o código a seguir.

```
01.     public static void main(String[] args) {
02.       //Obtém LocalDateTime trazendo o horário atual
03.       LocalDateTime horario = LocalDateTime.now();
04.
05.       System.out.println("LocalDateTime antes: " +
   horario);
06.       DateTimeFormatter formatador =
   DateTimeFormatter.ofPattern("HH:mm:ss");
07.       String horarioFormatado = agora.format(formatador);
08.       System.out.println("LocalDateTime depois: " +
   horarioFormatado);
09.
10.     }
```

Desafio: combine os tópicos abordados nesta aula e construa um *HashMap* que irá mapear nomes (*string*) com seus respectivos aniversários representados por um *LocalDate*. Registre ao menos três aniversários no mapa e depois, utilizando um *loop*, imprima todos os aniversários.

FINALIZANDO

Nesta aula, abordamos diversos assuntos, com especial destaque à visibilidade e ao encapsulamento. Esses conceitos são de extrema importância para a abstração dentro da orientação

a objetos, permitindo esconder detalhes de implementação e tornar classes complexas em termos de implementação simples de serem utilizadas.

Também discutimos *Collections, iterators* e a classe *LocalDate,* tópicos importantes para quem deseja aprofundar o desenvolvimento em Java. Outras linguagens de programação orientada a objetos possuem classes semelhantes para representar estruturas de dados iteradores e *data*, portanto, apesar de particular à linguagem Java, o conceito acaba se tornando amplo na prática. Nas próximas aulas, continuaremos com o conteúdo de *herança*, outro importante pilar da orientação a objetos.

REFERÊNCIAS

BARNES, D. J.; KÖLLING, M. **Programação orientada a objetos com Java**. 4. ed. São Paulo: Pearson Prentice Hall, 2009.

DEITEL, P.; DEITEL, H. **Java**: como programar. 10. ed. São Paulo: Pearson, 2017

LARMAN, C. **Utilizando UML e padrões**: uma introdução à análise e ao projeto orientados a objetos e ao desenvolvimento iterativo. 3. ed. Porto Alegre: Bookman, 2007.

MEDEIROS, E. S. de. **Desenvolvendo *software* com UML 2.0**: definitivo. São Paulo: Pearson Makron Books, 2004.

PAGE-JONES, M. **Fundamentos do desenho orientado a objetos com UML**. São Paulo: Makron Book, 2001

PFLEEGER, S. L. **Engenharia de *software***: teoria e prática. 2. ed. São Paulo: Prentice Hall, 2004.

SINTES, T. **Aprenda programação orientada a objetos em 21 dias**. 5. reimpressão. São Paulo: Pearson Education do Brasil, 2014.

SOMMERVILLE, I. **Engenharia de *software***. 9. ed. São Paulo: Pearson, 2011.