# PROGRAMAÇÃO ORIENTADA A OBJETOS

## CAPÍTULO 2 – COMO GARANTIR A SEGURANÇA DOS OBJETOS?

Handerson Bezerra Medeiros

# Introdução

Quando trabalhamos com Programação Orientada a Objetos, sabemos que vamos seguir um paradigma no qual criamos uma lógica computacional modelada em um problema do mundo real. Para isso, precisamos identificar nossos objetos, modelar as classes que definem a estrutura destes objetos, criar seus atributos e elaborar seus comportamentos (métodos). A partir desses métodos, conseguimos identificar como esses objetos podem se relacionar.

Com isso, chegamos à conclusão de que um programa nada mais é do que arranjos (algoritmos) dos objetos, que se relacionam com outros objetos, por meio de mensagens (métodos), com o intuito de alcançar uma solução para um problema. E para alcançar a melhor solução, precisamos fazer todo esse planejamento de maneira segura. Mas como realizar esse plano? Será que existe alguma maneira de melhorar nosso código, de forma a deixá-lo mais seguro?

Proteger nosso código significa melhorar a qualidade da produção do nosso programa. Ou seja, para realizarmos isso, precisamos identificar o que deve ser público (visível) em todo o código e o que deve ser escondido do código, salvo aquela classe à qual este código pertence. Porém, devemos lembrar que, mesmo o que for visível ou não, deverá ser possível de se comunicar. Parece difícil, mas o importante é lembrar que proteger o código, não significa separar nosso programa em partes.

Neste capítulo, vamos aprender como deixar nosso código protegido. Veremos o que significa encapsular o código e como podemos fazer isso. Consequentemente, poderemos verificar como melhoramos nossas classes e objetos. Vamos conhecer os principais tipos de coleções que um objeto pode possuir. Por fim, vamos começar a fazer associações das nossas classes, de forma que nossos objetos possam conversar entre si.

Vamos estudar esse conteúdo a partir de agora, acompanhe!

## 2.1 Mas... Por que encapsular?

Sabemos que encapsular significa esconder (abstrair) detalhes do código durante a implementação, porém, não podemos confundir encapsulamento com abstração. Sabemos que a programação orientada a objeto faz uma abstração do mundo real para o código. Então qual seria o real significado de encapsulamento? Vamos entender!

### *2.1.1 Encapsulando*

Para encapsular, devemos esconder todos os membros das nossas classes, e também esconder a forma como nosso código funciona (rotinas, métodos, etc.).

Mas por que esconder todas essas informações? Quando encapsulamos nosso código, construímos um programa mais suscetível a mudanças, ou seja, se for necessário fazer qualquer tipo de alteração na nossa lógica, em vez de mudarmos informações em vários lugares do código, iremos apenas fazer a alteração necessária em um único lugar.

Não entendeu? Vamos analisar um exemplo. Temos que construir um programa de sistema bancário, então, após nossa análise de requisitos, verificamos que é necessário que nosso programa tenha os métodos de sacar, depositar e transferir. Como já sabemos, precisamos construir os métodos responsáveis por essas ações (sacar, depositar e transferir). Porém, será que é importante e seguro que nosso usuário e outras classes do nosso sistema, tenha acesso aos atributos e métodos de forma a poder modicar o código? Por questão de praticidade e segurança, nós não devemos permitir esse tipo de acesso.

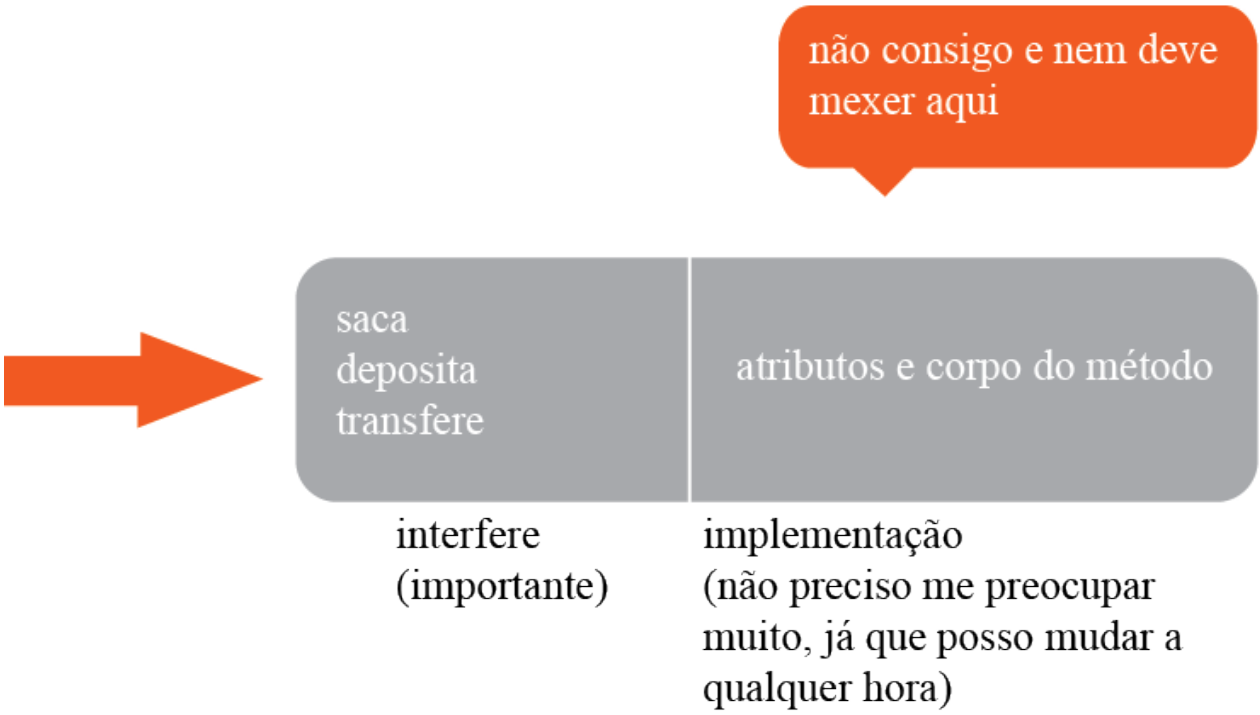


Figura 1 - Modelo da estrutura dos métodos de sacar, depositar e transferir de acordo com o que deve ser encapsulado.
Fonte: CAELUM, 2018, p. 65.

Como percebemos, encapsular não significa ocultar todo o código e, sim, ocultar apenas alguns elementos importantes para o desempenho do nosso programa, de uma classe das demais classes.

## VOCÊ SABIA?

Existe uma linguagem de programação que é muito confundida com a linguagem Java. Seu nome é Javascript, uma linguagem utilizada apenas nos *browsers* que, às vezes é confundida como sendo uma versão simplificada da linguagem Java. Diferente do Java o Javascript não cria *applets,* ou aplicações independentes, ele oferece interatividade para páginas *web* (MOZILLA, 2016).

Segundo Manzano (2014), quando utilizamos da técnica de encapsulamento, estamos adicionando uma proteção ao redor dos atributos e, para termos acesso a esses atributos, basta criar métodos que liberam o acesso. Assim, estamos protegendo nosso programa contra qualquer efeito colateral indesejado, que poderia afetar esses atributos de forma indesejada e consequentemente modificando seus valores de forma inesperada.
Conseguimos prover o encapsulamento declarando que temos total controle de acesso a esses atributos. Mas como podemos fazer isso? Vamos utilizar modificadores de acesso, acompanhe a seguir.

### *2.1.2 Modificadores de acesso*

Vamos analisar a seguinte situação: precisamos desenvolver um programa que manuseia informações sobre seres vivos, como por exemplo, homem e cachorro. Como já sabemos, podemos identificar os dois seres vivos

como objetos e suas características como atributos. Porém, sabemos que podem existir atributos comuns entre esses dois objetos, por exemplo: nome, idade e peso. Como também podem existir atributos que são específicos para cada um desses objetos, o objeto *homem* pode conter um atributo chamado *cpf* e o objeto *cachorro* pode conter um atributo chamado *raça*.

A partir daí, podemos modelar os métodos necessários para cada objeto. Da mesma forma que os atributos, podem existir métodos em comum (andar), como também métodos distintos para cada objeto (homem – falar, e cachorro – latir).

De forma a evitar conflito nas informações, nós devemos encapsular esses atributos e métodos de forma que apenas a classe correspondendo àquele objeto pode manuseá-la. Então seu acesso ocorre por meio de uma interface. Como assim?

Fazendo uma analogia com o mundo real, vamos imaginar que estamos dirigindo um carro. Para isso, utilizamos sua interface (pedais, volante e marcha). Precisamos diretamente manusear o motor do nosso carro? Na verdade, nem precisamos aprender sobre o motor, que segundo a nossa analogia seria a implementação do carro. Agora imagine que trocamos de carro, será necessário aprendemos novamente a dirigir? Ou seja, para utilizar nosso programa (dirigir um carro) precisamos apenas saber o que ele faz e não como ele faz. Ficamos livre de mudar e modificar nossos métodos, de forma que não precisamos alterar sua interface.

Precisamos interagir com nosso código de maneira correta, e por isso, utilizamos modificadores de acessos. Para entendermos quais são esses modificadores e como utilizar, vamos visualizar o código abaixo. O objetivo do nosso programa é permitir ao usuário utilizar um sistema bancário. Inicialmente, identificamos os atributos *numeroDaConta*, *saldoDisponivel* e *nomeDoCliente* (que é do tipo *Clientes*). Após isso, criamos os métodos de *Sacar* e *Depositar*, ambos recebendo, por parâmetro, o valor a ser sacado ou depositado.

```
class ContaCorrente{
public Clientes nome;
public numeroDaConta;
public int saldoDisponivel;
public void Sacar(double valor){
this.saldo = saldo – valor;
}
public void Depositar(double valor){
this.saldo = saldo + valor;
}
}
```

Continuando nossa análise, vamos imaginar que o cliente deseja realizar um saque e um depósito em sua conta. Nosso programa realiza as seguintes instruções:

```
contaCorrente conta = new contaCorrente(); // Instanciando um objeto
conta.Sacar(500.0);
contaDepositar(300.0);
```

Porém, de acordo como o código foi implementado, pode ser realizada uma instrução errônea, por exemplo, acesso direto aos atributos:

```
conta.saldoDisponivel = saldoDisponivel – 500.0;
conta.saldoDisponivel = saldoDisponivel + 300.0;
```

Aparentemente, pode não transparecer que existe um erro. Mas, imagine agora que a cada transação realizada (saque ou depósito), o banco vai cobrar uma taxa de serviço. Deixar a possibilidade de alteração do valor armazenado no atributo *saldoDisponivel,* ser realizado de maneira direita, significa fazer alteração no código toda vez que for realizada uma instrução de saque ou depósito. Isso deixaria nosso desenvolvimento bem lento e de difícil manutenção, correto? Então, para prevenir isso, devemos realizar o encapsulamento de algumas informações de forma que precisamos, nessa situação, realizar apenas alteração nos métodos de *Sacar* e *Depositar*.

Neste exemplo de sistema bancário, este problema acontece, pois definimos todos os atributos e métodos como sendo do tipo de acesso *Public*. Quando fazemos isso, estamos informando ao nosso programa, que todos os atributos e métodos dessa classe podem ser acessados de maneira direta por qualquer lugar do programa, ou seja, qualquer classe pode acessar e manipular aqueles elementos identificados como *public*.

## VOCÊ O CONHECE?

A linguagem de programa Java conta com a presença de uma mascote, de nome Duke. Sua história teve origem na Sun Microsystems, quando iniciaram o projeto *Green*, liderado por três pesquisadores. Duke foi criada por um desses pesquisadores, Joe Palrang. Assim que lançaram a tecnologia Java, por algum motivo (até então desconhecido), lançaram junto a mascote Duke. Alguns especulam sua criação, e acreditam que sua forma foi inspirada no emblema dos tripulantes da nave Enterprise, da série Star Trek (ORACLE, 2018b).

Uma forma de melhorar o acesso a essas informações, seria utilizar o modificador de acesso chamado *Private*. Quando optamos pelo uso do *private*, deixamos esses atributos visíveis apenas para a classe na qual eles estão declarados Voltando para o exemplo da nossa classe *contaCorrente*, podemos utilizar *private* em todos os seus atributos. Como estamos apenas alterando os atributos, significa que os métodos *sacar* e *depositar* ainda podem ser acessados pois estão declarados como *public*.

*private Clientes nome;*
*private numeroDaConta;*
*private int saldoDisponivel;*

Porém, as vezes precisamos deixar alguns atributos visíveis para algumas classes poderem utilizá-la. Essas classes ou subclasses devem pertencer ao mesmo pacote do meu programa. Para realizarmos essa opção, precisamos definir este atributo como sendo do tipo *protected*. Por exemplo, se quisermos que o atributo nome fique disponível para outras classes, definimos:

*protected Clientes nome;*

Podemos também, construir nosso programa sem definir modificadores de acesso. Por padrão o código vai liberar o acesso dessas informações apenas ao pacote no qual ele se encontra. Vamos resumir nossos conceitos? Veja o quadro a seguir.

| Modificador | Classe | Pacote | Subclasse | Globalmente |
|---|---|---|---|---|
| *Public* | Sim | Sim | Sim | Sim |
| *Protected* | Sim | Sim | Sim | Não |
| *Private* | Sim | Não | Não | Não |
| **Sem Modificador** | Sim | Sim | Não | Não |

Figura 2 - Tipos de modificadores de acesso com suas respectivas liberações no sistema.
Fonte: Elaborado pelo autor, adaptado de ORACLE, 2018a.

Como podemos perceber, precisamos identificar qual modificador de acesso deverá ter nosso programa, de forma que ele fique com o melhor desenvolvimento possível, garantindo assim, um programa com mais qualidade.

### *2.1.3 Implementando classes encapsuladas*

Na hora da implementação de classes encapsuladas, precisamos ter bastante cuidado sobre qual modificador de acesso utilizar. Lembra do nosso sistema bancário? Se identificarmos o atributo *numeroDaConta* como *private,* podemos causar problemas durante a implementação. Por exemplo, como poderemos acessar essa informação para verificar se a conta foi acessada com sucesso?
Diferente do que você pode estar pensando, não resolvemos o problema atribuindo *public* a este atributo, pois estaríamos ocasionando outros problemas, que já mencionamos anteriormente.
Uma forma de resolver isso, seria criar métodos de acesso para leitura e atribuição de valores a esses atributos. Essas propriedades são chamadas de *get*, para executar a leitura do atributo e *set*, para a escrita de um valor no atributo. Se quisermos acessar o atributo *numeroDaConta*, podemos realizar os seguintes métodos:

```
private int numeroDaConta;
public getNumeroDaConta(){ //código para ler o atributo
return numeroDaConta;
}
public setNumeroDaConta(int numero){ //código para escrever no atributo
this.numeroDaConta = numero;
}
```

É importante lembrar que o método *set* deve estar acessível em casos especiais (por exemplo, quando o valor passado precisa ser verificado), pois se um atributo possui *get* e *set*, nada o difere de deixarmos o atributo como *public*.
Agora, se o usuário desejar saber o número da conta, poderá chamar o método *getNumeroDaConta*. E caso queira escrever um valor ou alterar o valor existente da conta deverá utilizar o método *setNumeroDaConta*, passando por parâmetro o valor que será atribuído ao atributo. Simples, né? Vamos continuar melhorando nossas classes e objetos?

## 2.2 Melhorando nossas classes e objetos

Até agora, nossas classes e objetos parecem funcionar sem nenhum problema. Mas será que podemos melhorar nossa implementação mais um pouco?
Vamos imaginar o seguinte cenário: precisamos desenvolver um programa que controla o cadastro de clientes de uma loja.
Pelo que já sabemos, podemos criar uma classe chamada *Pessoas* que será a estrutura básica dos atributos e métodos disponíveis para um cliente. Criando nossa classe, temos:

```
class Pessoas{
private int contato;
private string nomeCliente;
//Métodos
//...
}
```

Porém, nosso sistema precisa registrar a quantidade de clientes que estão sendo armazenados. Bom, podemos imaginar, como solução, o seguinte: toda vez que for instanciado um novo cliente, o sistema imediatamente guardará em uma variável a soma de mais um cliente no sistema. Por exemplo,

```
Pessoas cliente = new Pessoas();
```

*totalDeClientes = totalDeClientes +1;*

De certa forma, essa solução parece resolver nosso problema, correto? Mas vamos analisar o seguinte: será que se precisarmos fazer alguma alteração nessa regra de negócio do meu programa, temos que procurar por toda a aplicação na qual pode ter essa linha de instrução e modificá-la uma por vez? Concorda que isso dá um trabalho imenso? Se por acaso esquecemos qualquer instrução, podemos gerar um imenso problema para o nosso programa. Como podemos resolver isso?

Sabemos que os construtores na nossa classe são métodos que são executados no momento em que é instanciado um novo objeto pertencente àquela classe. Então, será que se adicionar essa instrução de total de cliente toda vez que for instanciado um novo objeto, dará certo?

```
class Pessoa{
private int contato;
private string nomeCliente;
private int totalDeClientes;
Pessoa(){
this.totalDeClientes = this.totalDeClientes +1;
}
}
```

Como um objeto possui todos os atributos da sua classe, qual seria o valor do atributo *totalDeClientes*, se instanciarmos dois objetos do tipo *Pessoa*?

```
Pessoa cliente1 = new Pessoa();
Pessoa cliente2 = new Pessoa();
```

Por esses objetos serem do mesmo tipo, significa que eles compartilham os mesmos atributos, ou seja, para cada objeto do tipo classe criado, o atributo *totalDeClientes* teria o valor 1 armazenado. Pois esse atributo seria de cada objeto. E agora? Será que podemos criar uma variável que armazene essa informação e que seja compartilhada por todos os objetos?

Quando definimos que o atributo é único, significa que, quando o valor for alterado em um objeto, todos os outros objetos que compartilham daquele atributo teriam seu valor naquela variável alterada também. Como fazemos isso? Em Java, podemos declarar essa variável como sendo do tipo estática (*static*). Fazendo o uso do tipo *static*, nosso atributo se torna um atributo da classe, e não um atributo para cada objeto do tipo daquela classe. Vamos verificar como ficaria nossa classe *Pessoas* fazendo o uso do *static*.

```
class Pessoa{
private int contato;
private String nomeCliente;
private static int totalDeClientes; //Atributo da classe
Pessoa(){
Pessoa.totalDeClientes = Pessoa.totalDeClientes +1;
}
}
```

Como podemos perceber, por agora, o atributo *totalDeClientes* ser um atributo da classe, não colocamos o *this*. *totalDeClientes*. Dessa forma, precisamos informar que essa instrução deve ser realizada em todos os objetos, então modificamos esta instrução por *Pessoa.totalDeClientes* (podemos utilizar apenas a instrução *totalDeClientes*, porém, já vimos que utilizando a identificação *this* (que nesse caso é *Pessoa*) antes do atributo reforçaremos que esse atributo é do objeto).

Analisando nosso código, percebemos que identificamos esse atributo com o modificar de acesso *private*. Dessa forma, esse atributo só é visto pela própria classe, correto? Mas, para ser possível a todos os objetos deste, tipo enxergarem esse atributo, precisamos identificá-lo como *public*?

Se fizermos isso, podemos gerar uma possível falha no nosso programa, pois deixaremos esse atributo visível para todo o programa. Para resolver isso, basta criarmos um método *get* para a leitura deste atributo.

```
class Pessoa{
```

*private int contato;*
*private string nomeCliente;*
*private static int totalDeClientes; //Atributo da classe*
*Pessoa(){*
*Pessoa.totalDeClientes = Pessoa.totalDeClientes +1;*
*}*
*public int getTotalDeClientes(){*
*return Pessoa.totalDeClientes;*
*}*
*}*
Se agora nosso programa precisar dessa informação, basta apenas solicitar. Fazendo da forma que aprendemos anteriormente, significa que precisamos instanciar um objeto primeiro, para depois ter acesso aos métodos da classe, correto?
*Pessoa cliente = new Pessoa();*
*int totalPessoas = cliente.getTotalDeClientes();*

## VOCÊ QUER LER?

O Java possui um conjunto de ferramentas baseado para desenvolvedores. Esse conjunto de ferramentas é conhecido como Java *Development Kit* ou simplesmente JDK. Para desenvolver nossos programas em Java, se faz necessária a instalação dessas ferramentas, independente da IDE que você escolher usar, por exemplo, Eclipse ou NetBeans (PALMEIRA, 2012). Quer entender melhor como funciona esse conjunto de ferramentas? Saber o que temos disponível em nossa programação? Leia: <https://www.devmedia.com.br/entendendo-e-conhecendo-as-versoes-do-java/25210>.

Vamos pensar um pouco. E se não tivesse nenhum cliente cadastrado ainda, significa que não teríamos acesso a essa informação? Pois, da forma que colocamos nossas instruções acima, é obrigatório instanciar, pelo menos, um novo cliente para poder ter acesso ao atributo *get.TotalDeClientes*. Isso pode gerar erro em nosso programa. Resolvemos isso de uma maneira bem simples. Da mesma forma que transformamos o atributo *totalDeClientes* estático, para que ele tenha acesso em todos os objetos, podemos transformar o nosso método *getTotalDeClientes* em estático também.
*public static int getTotalDeClientes(){*
*return Pessoa.totalDeClientes;*
*}*
Dessa forma, ao acessarmos essa informação, precisamos apenas chamar esse método atrás da sua classe, retornando seu valor em uma variável qualquer e não por meio de uma referência de seu objeto.
*int totalPessoas = Pessoa.getTotalDeClientes();*
Para não restar dúvidas, vamos analisar o seguinte problema: precisamos desenvolver um programa que guarde o registro de carros de uma concessionária. Além de identificar as características de cada carro, é importante que o gerente tenha o conhecimento de quantos carros existem disponíveis em estoque.
Antes de qualquer coisa, precisamos modelar nossa classe, que receberá os atributos e métodos que são compartilhados pelos objetos. Vamos chamar nossa classe de Carro.
*public class Carro{*
*//Atributos*

```
private string marca;
private int anoModelo;
public static int quantidadeDeCarros;
//Construtor
Carro(){
Carro.quantidadeDeCarros = Carro.quantidadeDeCarros +1;
}
//Métodos de leitura e escrita para os atributos private
public void setMarca (string marca){
this.marca = marca;
}
public void setAnoModelo (int ano){
this.anoModelo = ano;
}
public string getMarca (){
return this.marca;
}
public int getAnoModelo (){
return this.anoModelo;
}
//Método de leitura para o atributo static
public static int getQuantidadeDeCarros(){
return Carros.quantidadeDeCarros;
}
}
```

Com a utilização da estrutura acima, quando nosso programa precisar saber a quantidade de carros em estoque, poderá fazer uma chamada direta ao método *getQuantidadeDeCarros*. Percebemos que, utilizando variáveis estáticas no nosso programa, podemos controlar, por exemplo, o número total de objetos de uma classe, como também compartilhar determinada informação entre objetos do mesmo tipo.

## 2.3 Conhecendo as coleções de objetos

Para a melhorar nosso desenvolvimento, podemos fazer o uso de uma API em Java, chamada Java *Colletions*. Mas o que significa isso? Quando utilizamos coleção, podemos agrupar um conjunto de objetos de forma que podemos desenvolver algumas operações básicas nessa coleção.

Deitel e Deitel (2016) afirmam que, utilizando o conceito de coleções em nosso código, podemos utilizar uma estrutura de dados já existentes (embutida pela linguagem de programação), dessa forma não precisamos nos preocupar com o modo como será implementada essa instrução.

A API *collections* disponibiliza uma estrutura dinâmica para utilização. Na figura a seguir, podemos verificar sua expansão. É importante lembrar que, para a linguagem Java, as coleções também são objetos, ou seja, seguem uma estrutura definida por sua classe, e podem interagir com outros objetos.

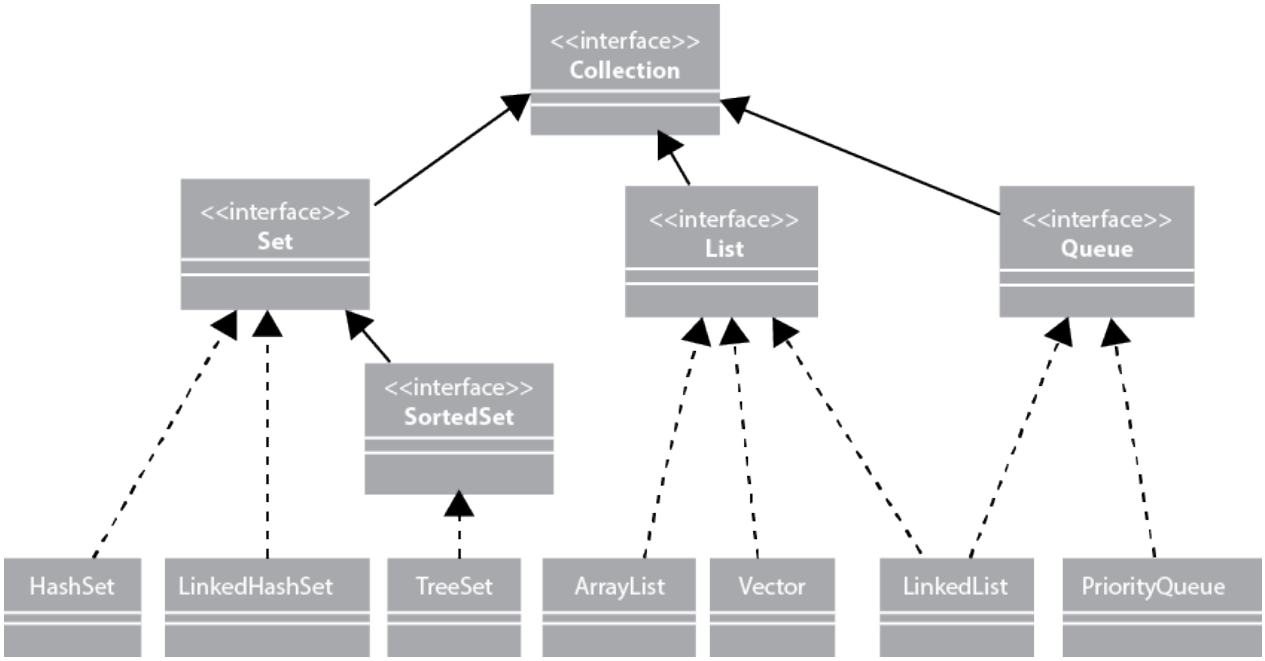


Figura 3 - Árvore da hierarquia de interfaces e classes da Java Collections Framework que são derivadas da interface Collection.
Fonte: Elaborada pelo autor, adaptada de ZUKOWSKI, 2001, p. 5.

Como podemos notar, *Collections Framework* é um conjunto bem definido de interfaces e classes para representar e tratar grupos de dados como uma única unidade (PALMEIRA, 2018). Para entender melhor, vamos dividir a coleção em três elementos. Podemos citar como primeiro elemento, a interface.

Interfaces são tipos abstratos que representam as coleções. Na Figura acima podemos notar como é sua hierarquia de interface. Quando programamos com uma interface (por exemplo, *Set* e *List*) significa que o seu acesso deverá ser feito obrigatoriamente pelos objetos, apenas para uso dos seus métodos. O segundo elemento trata das Implementações concretas das interfaces, por exemplo, *HashSet* e *ArrayList*. Por fim, temos o algoritmo, que simplesmente são os métodos que estão disponíveis pela interface para ser utilizada pelos objetos.

Existe ainda, uma outra classe, que implementa uma interface chamada MAP. Apesar de não ser considerada uma *Collection*, a interface *Map* faz parte da API *Collection*. Portanto, tem o mesmo conceito de coleções e pode ser utilizada em nossos códigos. A figura a seguir apresenta a estrutura da interface *Map*, mostrando que seria sua interface e suas implementações.

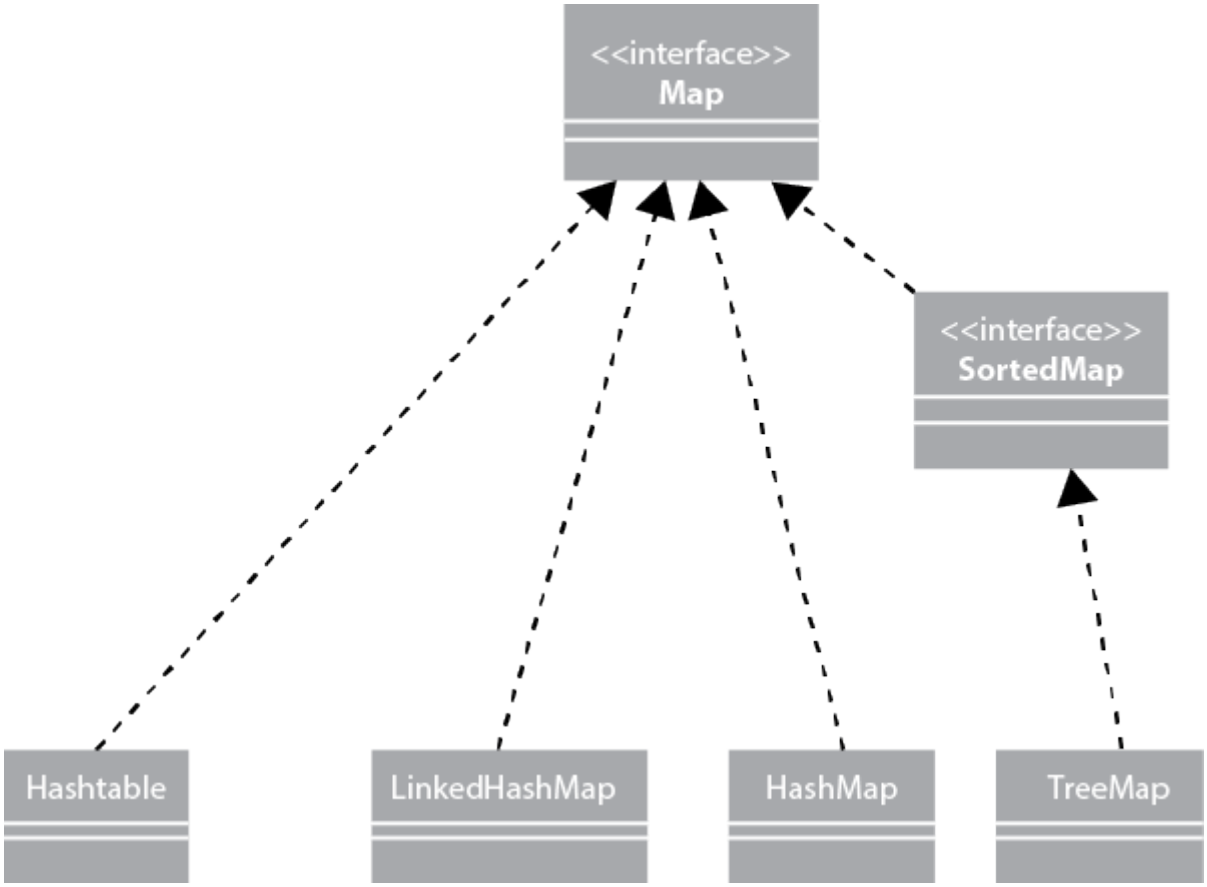


Figura 4 - Árvore da hierarquia de interface Map que também faz parte da API Java Collections Framework.
Fonte: Elaborada pelo autor, adaptada de ZUKOWSKI, 2001, p. 5.

Ainda não entendeu qual é a vantagem de utilizar coleções? Quando criamos coleções podemos ter acesso a métodos já implementados, na maioria das interfaces. Por exemplo, métodos para adicionar ou remover um objeto dentro da coleção e pesquisar algum objeto específico dentro da coleção, a partir de alguma característica. Podemos utilizar coleções organizadas ou ordenadas. Mas isso não significa a mesma coisa? Não, quando falamos que a coleção está organizada, significa que a cada inserção de um novo objeto, este será inserido no final da fila. Já se utilizarmos coleções ordenadas, cada novo objeto inserido será ordenado na fila, de acordo com alguma instrução, por exemplo, do menor para o maior. Bem útil, concorda? Para saber o comportamento das implementações disponíveis no *collection*, dá uma olhada no quadro a seguir.

| Organizada | Não organizada | Ordenada | Não ordenada |
|---|---|---|---|
| *LinkedHashSet, ArrayList, Vector, LinkedList, LinkedHashMap* | *HashSet, TreeSet, PriorityQueue, HashMap, HashTable, TreeMap* | *TreeSet, PriorityQueue, TreeMap* | *HashSet, LinkedHashSet, ArrayList, Vector, LinkedList, HashMap, HashTable, LinkedHashMap* |

Quadro 1 - Comportamento de ordenação e organização das implementações disponíveis na API Java Collections Framework.
Fonte: Elaborado pelo autor, 2018.

Temos ainda mais algumas funcionalidades da API Collections que podemos utilizar junto com as coleções em nosso programa. Existem duas interfaces disponíveis, que tem como objetivo, percorrer qualquer coleção de objetos que seja derivada de *Collection*. Uma dessas interfaces é a *Iterator*, que, quando utilizada, nos ajuda a percorrer uma coleção e remover algum elemento. Vamos colocar em prática essa interface, logo mais, neste capítulo.

# VOCÊ QUER VER?

Você já deve ter percebido que a API *Collections* nos oferece muitas opções de implementações. Quando sabemos utilizá-las, construímos nosso código de maneira mais eficaz e com qualidade. O grande desafio da *collections* é implementar diversos tipos possíveis de coleções com suas respectivas características peculiares. No vídeo <https://www.youtube.com/watch?v=m6JNiSY8rrE>, você pode aprender um pouco mais sobre a hierarquia do *collections* (PAGLIARES, 2012).

Já sabemos o que são coleções e por que devemos utilizá-las, mas como funciona cada implementação? Precisamos entender como cada uma delas funciona para escolher qual melhor se adapta aos nossos problemas.

## *2.3.1 Tipos de coleções e suas implementações*

Percebemos que a variedade de opções de tipos de coleções é bem considerável. E agora? Para escolher a melhor opção para nosso problema, precisamos estudar um pouco mais sobre cada tipo existente.
É necessário identificar quais operações básicas vamos utilizar em nosso código, pois algumas operações (adição, remoção, acesso e pesquisa) podem obter um melhor desempenho dependendo do tipo de coleção, como também podem possuir algum tipo de restrição ou funcionalidade especial. Parece difícil escolher? É só saber identificar. Vamos começar.

### Interface Collection

Como vimos na Figura da Árvore da hierarquia de interfaces e classes da Java *Collections Framework* ,que são derivadas da interface *Collection*, a interface *collection* é apresentada no topo da hierarquia. Sua responsabilidade é definir as operações básicas (adição, remoção, acesso e pesquisa) para as outras interfaces que estendem dela.
É importante deixar claro, que, na interface *collection* não é possível ter uma implementação direta. Ela simplesmente é utilizada pelas outras interfaces.

### Interface Set

Utilizamos a interface *set*, quando queremos definir uma coleção de objetos que não possua elementos duplicados. Podemos implementar essa interface pelo uso das implementações *HashSet*, *TreeSet*, *LinkedHashSet*.
No *HashSet*, nossas informações, não serão necessariamente adicionadas de maneira ordenada, pois ele guarda o elemento sem seguir uma ordem lógica. Ou seja, a ordem de saída não é de acordo com a ordem de inserção dos elementos. Porém, essa implementação é a mais rápida entre a interface *Set*, quando queremos ter acesso às informações. Devemos optar por essa implementação quando nosso programa precisar ter elementos que não sejam repetidos e cuja ordem de armazenamento não seja importante. Sua sintaxe na hora da declaração deverá apresentar informações de nome do objeto declarado e o tipo de objeto.
*HashSet<Nome> set = new Type<Nome>();*

Vamos supor que temos uma classe chamada *Pessoas*, que armazena objetos que referenciam no mundo real, os clientes de determinada loja. Vamos utilizar a estrutura de *HashSet* para adicionar clientes novos na coleção, comparando se aquele cliente já existe na coleção. Após isso, nosso programa exibirá a quantidade de clientes existentes e depois suas informações, e, para isso, podemos utilizar o auxílio da interface *iterator* comentada anteriormente neste capítulo. Nosso programa principal poderia ser implementado na estrutura abaixo.

```
public static void main(String[] args) {
HashSet<Pessoas> cliente = new HashSet<Pessoas>();
//adicionando com o auxílio dos métodos de Collections
cliente.add(new Pessoas("João");
cliente.add(new Pessoas ("Maria");
//criando mais um cliente sem o auxílio de collections
Pessoas outroCliente= new Pessoas"Maria");
//Verificação se já existe esse novo cliente na coleção cliente
if(cliente.contains(outroCliente)){
System.out.println("Cliente já cadastrado!");
}
//Verificar e imprimir tamanho da coleção
System.out.println("Tamanho coleção é: "+ cliente.size());
//Percorrendo a coleção e imprimindo as informações
Iterator< Pessoas > inf = cliente.iterator();
while(inf.hasNext()){
Pessoas infCliente = (Pessoas) inf.next();
System.out.println(infCliente);
}
}
```

Se optarmos pela utilização da implementação *TreeSet*, nossos dados serão classificados (devido à interface *SortedSet*), porém o acesso a eles será mais lento do que se utilizarmos o *HashSet*. Ou seja, quando precisarmos utilizar uma coleção que não possua elementos duplicados e que sua ordem de leitura siga a mesma ordem que os elementos foram inseridos, nossa melhor opção é implementarmos com *TreeSet*.

Por fim, temos o *LinkedHashSet*, basicamente seria um meio termo entre as outras duas implementações da interface *Set* (*HashSet* e *TreeSet*). É uma implementação derivada de HashSet. Seus objetos são iterados na mesma ordem que foram inseridos na aplicação. Como optar entre esses três métodos de implementação? Como podemos perceber a principal diferença entre eles se dá pela velocidade de execução e opção de ordenação.

## Interface List

Diferente da interface *Set*, usamos a interface *List* quando queremos que nossa coleção seja obrigatoriamente ordenada e que possa adicionar objetos duplicados. Se os objetos são inseridos de maneira sequencial, significa que o usuário vai poder ter controle sobre a posição de cada elemento, tendo em vista que eles devem ser inseridos por meio de um índice. Consequentemente, podemos acessar um objeto pelo seu índice, possibilitando assim um acesso a pontos específicos na coleção. Por se tratar de uma lista e esses objetos são inseridos em uma ordem linear, significa que cada objeto possui um antecessor (exceto o primeiro) e um sucessor (exceto o último). São implementações da interface *List*: *ArrayList*, *Vector* e *LinkedList*.

A implementação *ArrayList* nada mais é do que um *array* dinâmico. Você sabe o que significa *array*? É uma estrutura na qual os elementos são armazenados de uma maneira que eles possam ser identificados por um índice. Ou seja, a diferença entre o *array* dinâmico e convencional é que, no convencional, limitamos o seu tamanho, já no dinâmico, seu tamanho aumenta de acordo com a necessidade da coleção.

Por usarmos índices para identificar a localização dos elementos no *array*, a busca por elementos acaba se tornando bastante rápida. Mas já a inserção e remoção desses elementos não são rápidos, tendo em vista que sempre que optarmos por uma dessas opções (inserção ou remoção) nosso *array* deverá continuar ordenado. Vamos verificar a utilização do *ArrayList* na implementação do seguinte programa.

Precisamos catalogar os produtos existentes em um supermercado. Para isso, vamos adicionando no programa um produto por vez, porém, cada produto tem um código numérico que o identifica. Após inserir todos os códigos dos produtos presentes no estoque, o programa deverá mostrar ao usuário, os códigos dos produtos existentes de maneira ordenada. Por fim, podemos acessar um produto por meio de seu índice. É importante indicar para o *ArrayList* qual é o tipo de valor que ele vai receber, em nosso caso será do tipo inteiro, para que nesse *array* de inteiros não seja possível colocar um objeto *Pessoa*, assim evitando futuros erros.

```
public static void main(String args[]) {
ArrayList <Integer>produto = new ArrayList();
//Inserção dos produtos na coleção, informando por parâmetro o código
produto.add(12259);
produto.add(21);
produto.add(349);
produto.add(21);
produto.add(34798);
//Percorrendo a coleção e imprimindo as informações
Iterator inf = produto.iterator();
while(inf.hasNext()){
System.out.println(inf.next());
}
//Acessando por meio de um índice específico
System.out.println(produto.get(349));
}
```

Você concorda que pode haver mais de uma quantidade de um produto em estoque? Se são o mesmo produto, eles obrigatoriamente precisam ter o mesmo código de identificação, correto? Como podemos perceber, utilizar na implementação o *ArrayList* é uma opção para que os produtos que tenham o mesmo código, possam ser armazenados e contabilizados no estoque.

O *ArrayList* e o *Vector* são muito parecidos e, por vezes, acabam tendo o mesmo resultado. Por exemplo, no programa acima, sobre estoque do supermercado, se optarmos por utilizar *Vector,* o programa seria executado com sucesso. Aonde está a diferença entre eles?

Ambos têm a mesma funcionalidade de aumentar de tamanho dinamicamente, a diferença está em que o *ArrayList* aumenta em 50% do seu tamanho atual. Já o *Vector* aumenta seu tamanho o dobro, assim que sua lista está cheia. E agora, qual escolher?

Tudo vai depender do problema que você está querendo resolver computacionalmente. Se seu programa necessita aumentar a lista de tamanho com mais frequência, o melhor caso seria utilizar o *Vector*. Dessa forma não ocorreria problema com a performance do nosso programa, caso optássemos por *ArrayList* em vez do *Vector*

.

## VOCÊ SABIA?

A linguagem Java tem uma sintaxe parecida com outras linguagens de programação, por exemplo C e C++. Isso ocorre mesmo que elas possuam diferenças nos paradigmas, pois a linguagem C é baseada em programação estruturada, e C++, em orientação a objetos. Ou seja, após você aprender Java, você terá mais facilidade em aprender outras linguagens de programação (CESTA; RUBIRA, 2009).

Por fim, podemos optar também pela implementação *LinkedList*, que também pode ser implementada pela interface *Queue* (veremos logo mais à frente). O *LinkedList* é bastante parecido com as outras duas implementações da interface *List* (*ArrayList* e *Vector*). Sua principal diferença encontra-se nos seus métodos. Essa implementação conta com alguns métodos adicionais quando trabalhamos diretamente com o início ou final da lista, facilitando as operações de inserção, remoção e acesso.

Como podemos perceber, o uso das três implementações disponíveis pela interface *List* acabam executando as mesmas funções. O que pode dificultar a escolha entre elas? Porém, fica claro que precisamos compreender nosso problema, pois a principal diferença entre elas é sua performance na aplicação. Quando fazemos a melhor escolha, significa que estamos melhorando a performance do nosso programa.

## Interface Queue

Quando precisamos definir uma lista com prioridades, devemos optar pela interface *Queue*. Como assim uma lista com prioridades? Vamos analisar o seguinte cenário, quando estamos em uma fila de atendimento: como ocorre o processo de entrada de novas pessoas e a saída para ser atendido? Precisa haver uma prioridade, concorda? Então o primeiro a entrar na fila deverá obrigatoriamente ser o primeiro a sair da fila.

Vamos analisar outra situação: o entregador de *pizza*, quando vai organizar os pedidos de entrega, também deve seguir uma ordem de prioridades para empilhar as *pizzas* na mochila, correto? Nesse caso, o último a entrar na fila será o último a ser entregue. Compare a diferença entre fila e pilha na figura a seguir.

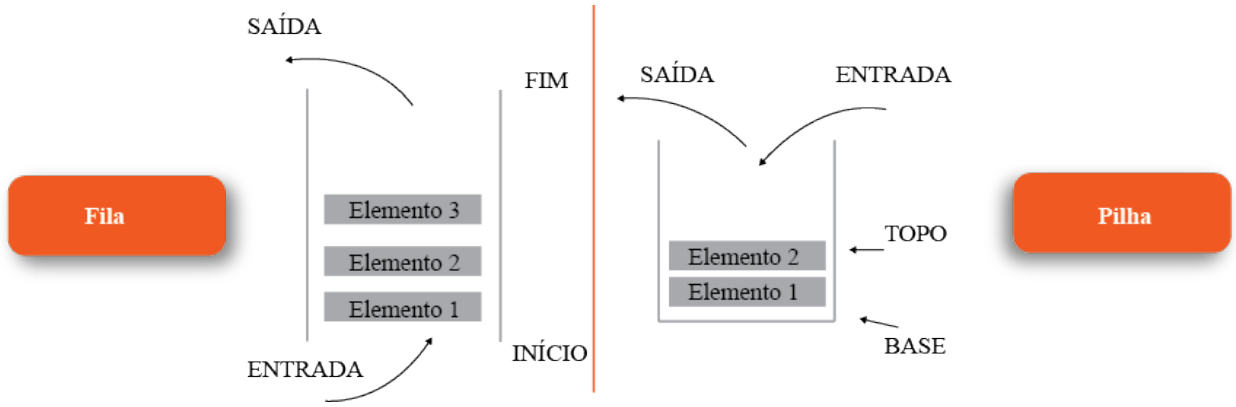


Figura 5 - Diferença ilustrativa do comportamento de inserção e remoção de elementos em uma fila e em uma pilha.

Fonte: Elaborada pelo autor, adaptada de MAIA, 2014.

Ou seja, a interface *Queue* vai ser utilizada quando a ordenação de inserção e remoção dos elementos forem importantes e aceitarem duplicação de elementos. Podemos implementar essa interface com o uso das implementações *PriorityQueue* e *LinkedList* (já comentada anteriormente nesse capítulo). Ambas opções vão ordenar, seguindo uma prioridade.

Por envolver ordenação, é necessário que a interface implemente duas interfaces chamadas *Comparable* e *Comparator*. A diferença na utilização entre *PriorityQueue* e *LinkedList* está ligada à utilização dessas outras interfaces.

Quando utilizamos a implementação *PriorityQueue*, o objeto que for inserido na coleção deverá implementar *Comparable* (usada quando existe uma ordem natural). Para isso precisamos implementá-la dentro do objeto pelo método *CompareTo*. Vamos desenvolver um programa para o qual os novos objetos *cliente* do tipo *Pessoas* deverão ser adicionados em uma fila de atendimento. E após isso, o programa deverá mostrar a ordem de chamada dos clientes para serem atendidos. Nosso programa principal ficaria:

```
public static void main(String[] args){
Queue cliente = new PriorityQueue();
```

```
//Instanciando novos cliente e adicionando na fila
cliente.offer(new Pessoas("Maria"));
cliente.offer(new Pessoas("João"));
//Enquanto estiver cliente em fila, imprima par ao usuário
while (cliente.size() > 0){
//Imprimindo objeto para o usuário.
System.out.println(cliente.poll());
}
}
```

Como estamos utilizando a implementação *PriorityQueue*, não podemos esquecer de modicar nossa classe *Pessoas* para que ela implemente a interface *Comparable* e possua um método de *compareTo*.

```
class Pessoas implements Comparable<Pessoas>{
public String nome;
public Pessoas (String nome){
this.nome = nome;
}
public int compareTo(Pessoas p){
return this.nome.compareTo(p.nome);
}
public String toString(){
return this.nome/
}
}
```

Quando utilizamos a implementação *LinkedList* junto com a interface *Queue*, significa que aquela coleção poderá implementar todas as operações de lista, permitindo a inserção de todos os elementos. Ou seja, diferente da implementação *PriorityQueue*, a implementação *LinkedList* aceita a inserção de elementos nulos na coleção.

## Interface Map

Para finalizar nosso assunto de coleções, temos a interface Map. Sua estrutura permite que cada elemento armazene duas informações: uma chave e um valor. Apesar dessa interface permitir duplicação de valores, as chaves não podem ser duplicadas. Por ser necessária a informação de chave e valor, na sintaxe de criação da interface deverão ser identificadas essas informações. Podemos implementar esta interface pelo uso das implementações *LinkedHashMap*, *HashMap* e *TreeMap*.

Dentre as implementações disponibilizadas pela interface *Map*, a implementação *HashMap* é a que mais se destaca no desenvolvimento. Suas principais características encontram-se em possuir uma estrutura na qual os elementos não são ordenados, possibilitando a inserção de valores e chaves nulas e fazendo com que as operações de busca e inserção seja rápida. Como exemplo, vamos pensar em um programa que necessite guardar uma lista de alunos. Sabemos que cada aluno possui uma identificação única (matrícula), mas também podem ter nomes iguais.

```
public static void main(String[] args){
Map<integer,String> alunos = new HashMap<integer,String>();
//Adicionando alunos
alunos.put(256, "João");
alunos.put(547, "Maria");
//Consultando se valor existe no HashMap
String nomeAluno = alunos.get(547);
//Removendo dados
Alunos.remove(547);
//Verificar se existe um aluno
Boolean verificar = alunos.containskey(256);
```

```
//Impressão para o usuário
System.out.println(nomeAluno);
if(verificar){
System.out.println("Aluno existe");
}
//Remover todos os itens existentes
alunos.clear();
}
```

Legal, né? O que difere, então, das outras implementações disponíveis na interface *Map*? A implementação *TreeMap* implementa uma outra interface chamada *SortedMap*. O que faz com que os objetos sejam adicionados de forma ascendente das chaves. Apesar de podemos personalizar essa ordem, é preferível que você utilize o *TreeMap* para mapa ordenado.

Já a implementação *LinkedHashMap* garante que a saída dos objetos seja realizada na mesma ordem de inserção. Resumindo, a diferença entre as implementações é referente à sua inserção e leitura dos objetos. No *HashMap* essa ordem é aleatória, no *TreeMap*, a ordem é obedecida por alguma informação de ordenação e no *LinkedHashMap*, a saída é de acordo com sua inserção.

## VOCÊ QUER LER?

Para um programa Java ser executado, é preciso que exista um compilador para realizar esse procedimento. Em Java, é utilizado um compilador chamado *byte-code*. Esse compilador transforma nosso código Java em uma linguagem mais entendível para a máquina (RICARTE, 2017). Quer saber mais um pouco sobre esse compilador? Conheça mais em: <http://www.dca.fee.unicamp.br/cursos/PooJava/javaenv/bytecode.html>.

Já sabemos como melhorar nossas classes utilizando coleções, mas como eu posso fazer para essas classes conseguirem interagir uma com as outras? Vamos aprender.

# 2.4 Interagindo com as classes

Aprendemos a pensar no problema, de forma a construir uma solução orientada a objetos. Para isso, modelamos as classes e criamos os objetos. Porém, para o programa funcionar, esses elementos precisam conversar entre si, interagindo de acordo com a lógica que implementamos. Mas como podemos realizar essa comunicação?

Sabendo que uma classe nada mais é do que a descrição de atributos e comportamentos de um tipo de objeto, este objeto precisa ser instanciado desta classe. Para acontecer essa comunicação de classes/objetos com outras classes/objetos, precisamos criar uma relação lógica entre eles. Podendo ser uma relação de associação, composição ou agregação. Vamos entender cada uma delas, a seguir.

## *2.4.1 Associação*

Vamos analisar o seguinte cenário, criamos uma classe chamada *CorpoHumano*, que contém atributos e comportamentos, que todos os objetos humanos devem possuir. Por exemplo: *membro1*, *membro2*, *membro3*, *membro4* (referente a braços e pernas), como atributos e *pegar* e *andar*, como métodos. Porém, para o nosso programa, se faz necessário guardar informações mais específicas sobre um elemento (membro). Sendo assim,

optamos por criar uma outra classe chamada *Membro*, que terá como atributo *força*, do objeto *membro*, e um método chamado *fazerForça*.
Você concorda que, neste cenário, nós temos uma classe que precisa ser associada junto com a outra classe? Veja, quando criamos um objeto *CorpoHumano*, obrigatoriamente, precisamos criar quatro objetos *Membros*, que precisam conversar com este objeto *CorpoHumano*.
De acordo com Camacho (2005), associação é uma conexão entre classes, consequentemente será uma conexão entre seus objetos também. Significa afirmar que uma classe precisa conhecer a outra.
Voltando para o cenário relatado acima, para cada objeto *CorpoHumano,* existem quatro objetos *Membros*, associados a ele. Para representar essa associação, precisamos identificar na classe *CorpoHumano*, que seu atributo membro é do tipo *Membro*, veja:

```
public class CorpoHumano{
//Declarando variável do tipo Membro
public Membro membro1 = new Membro;
public Membro membro2 = new Membro;
public Membro membro3 = new Membro;
public Membro membro4 = new Membro;
//Métodos da classe CorpoHumano
public voide pegar(){
System.out.println("Pegando!");
}
public void andar(){
System.out.println("Andando!");
}
}
public class Membro{
//Declarando variável
int forcaDoMembro;
//Metodo
public void fazerForca(){
System.out.println("Fazendo força com os membros!");
}
}
```

Simples, né? Quando nosso programa permite fazer associação, conseguimos separar melhor as informações e manuseá-las de maneira mais adequada.

## *2.4.2 Agregação*

Diferente da associação, a relação de agregação é realizada quando precisamos que uma classe tenha acesso a informações de outra classe, dentro de suas operações. Ficou confuso? Vamos analisar o seguinte problema: precisamos de um programa no qual seja possível realizar o pagamento aos jogadores e técnicos de uma partida. Este pagamento é diferenciado, caso haja gols na partida. Sabemos que, das partes envolvidas neste processo, temos: os gols que podem ter sido realizados, o jogador que participou da partida, o treinador que coordenou o time naquela partida e, por fim, o pagamento a ser realizado.
Para fins de melhorar nossa implementação, identificamos cada um desses elementos como sendo uma classe distinta, ou seja, criaremos uma classe *Gol*, classe *Treinador*, classe *Jogador* e classe *Pagamento_Partida*. Cada uma dessas classes possuirá os atributos e métodos que definem seus objetos. A princípio, vamos criar apenas as classes de *Gol*, *Treinador* e *Jogador*.

```
public class Gol{
public int gol;
```

```
public setGol(int valor)
this.gol = valor;
public getGol(){
return this.gol;
}
}
public class Treinador{
public String nome;
public float comissao;
public void setComissao(float valor){
this.comissao = valor;
}
}
public class JogadorCliente{
public String nome;
public float salario;
public void setSalario(float valor){
this.dinheiro = valor;
}
public void getSalario(){
return this.dinheiro;
}
}
```

Ate então, não temos nenhuma novidade em nossa implementação. Porém, vamos analisar um pouco a nossa classe de *Pagamento_PartidaVenda*, que ainda vamos criar. Essa classe ficará responsável por concretizar o pagamento de uma partida e, para isso, ela precisa das informações dos gols que foram realizados na partida, o jogador que participou da partida e do treinador que coordenou o time. Para realizar os pagamentos, o jogador tem seu salário fixo, mais uma percentagem de 10% na quantidade de gols realizados na partida.

Já o treinador não possui salário fixo, seu pagamento é feito com a comissão de 50% na quantidade de gols realizados. Ou seja, podemos afirmar que a classe *Pagamento_Partida* é composta por todas as outras classes que criamos. Então concluímos que a classe venda contém as classes *gol*, *jogador* e *treinador*.

```
public class Pagamento_Partida{
//Declarando classes que são do tipo de outras classes
public Gol gol = new Gol;
public Treinador treinador = new Treinador;
public Jogador jogador = new Jogador;
//Métodos que chamam métodos de outras classes
public void inserirGoldaPartida(){
gol.setGol(3);
}
public void realizarPagamento(){
System.out.println("Jogos realizados com sucesso!");
jogador. setDinheiro (jogador. getDinheiro() + (gol.getGol()*0.1);
treinador. setComissão(gol.getGol. * 0.5);
}
public void cancelarPagamento(){
System.out.println("Jogo não realizado!");
}
}
```

Como podemos perceber a agregação, na verdade, acaba sendo um pedaço da associação também.

### *2.4.3 Composição*

Por fim, temos nossa relação de composição. Da mesma forma que a relação de agregação, a comunicação ocorre quando uma classe utiliza uma parte de outras classes, dentro dos seus métodos, realizando operações. Então, qual seria a diferença entre agregação e composição?
Vamos lembrar de nosso programa de vendas, caso não ocorra uma venda, não significa que as outras classes (produto, vendedor e cliente) precisam deixar de existir. Ou seja, a principal diferença entre essas duas relações ocorre porque, na relação composição, se a classe que utiliza as outras classes, deixar de existir no programa, obrigatoriamente as outras classes, com as quais ela faz relação, deixam de existir também.
Vamos colocar em prática nossa relação de composição, analisando o seguinte problema. Precisamos criar um sistema bancário, no qual o banco deverá armazenar todas as contas-correntes e poupanças existentes. Como pode existir vários objetos do tipo conta-corrente e poupança, vamos criar classes que modelem esses dois tipos de contas diferentes. E por fim, uma classe *Banco*, que faz a manipulação dos métodos das classes *conta-corrente* e *poupança*.
Nossa implementação é realizada da mesma forma que fizemos na relação de agregação. A diferença está no modelo de negócio que estamos construindo. Pois, caso excluamos nossa classe *Banco*, as outras duas classes ficariam inutilizáveis. Pois somente a classe *Banco* manipula essas classes.

## CASO

A ideia do uso de orientação a objetos nos ajuda a pensar de forma diferente, já que seus paradigmas estão saindo do universo da computação, para outros campos de aplicação. Na área administrativa, já podemos utilizar esses conceitos como uma nova técnica para administrar os recursos de mão de obra das empresas, derivando os conceitos usados na técnica de orientação a objetos da engenharia de *software*.

A ideia se trata de quebrar com a ideia de hierarquia vertical, no topo está o presidente da empresa, e depois, cada divisão é composta por uma série de quadros intermédios e supervisores responsáveis por departamentos, passando a tratar a todos esses elementos de colaboradores sob a ótica de classes, objetos, atributos e métodos.

Dessa forma, um funcionário passaria a ser classificado na empresa pelas suas características técnicas, funcionais, teóricas, intelectuais e interpessoais. Tais características serão transformadas em recursos, para serem usados nos projetos da empresa (REGI JUNIOR, 2009).

Percebemos que temos disponíveis em Java, várias técnicas de melhorar nossa programação. Ao final, conseguimos garantir mais segurança em nosso código e um melhor desempenho em nosso programa.

# Síntese

Chegamos ao final do capítulo. Aprendemos a programar de maneira mais eficaz, expandindo, com as técnicas aprendidas, uma lógica melhor. Estudamos a importância de encapsular nossas informações, conhecemos novas

formas de criar métodos, identificando as possíveis coleções que podem ser utilizadas nos objetos e, por fim, verificando como fazemos essas classes conversarem.

Neste capítulo, você teve a oportunidade de:

- desenvolver classes encapsuladas, distinguindo quais modificadores devem ser utilizados, de acordo com o relacionamento;
- melhorar nossa segurança, criando atributos e métodos estáticos;
- conhecemos os tipos de coleções que podemos utilizar em nossa programação, verificando como funciona cada uma das interfaces e implementações;
- aplicamos relações de comunicação para possibilitar que nossas classes conversem entre si.

# Bibliografia


CAMACHO, R. **Algoritmos e estrutura de dados II**. LIACC/FEUP - Universidade do Porto, 2005. Disponível em: <https://web.fe.up.pt/~aed2/acetatos/aula6.pdf>. Acesso em: 15/08/2018.

CAELUM. **Java e Orientação a objetos**. Caelum, 2018. 65p.

CESTA, A. A; RUBIRA, C. M. F. **Tutorial**: a linguagem de programação Java e orientação a objetos. Instituto de Computação, Julho 1996; Atualizado em 2009. Campinas: Unicamp, 2009. Disponível em: <http://www.ic.unicamp.br/~cmrubira/JAVATUT14PDF.pdf>. Acesso em: 15/08/2018.

DEITEL, P; DEITEL, H. **Java**: como programar. 10. ed. São Paulo: Pearson: 2016.

MAIA, G. **Lista, Fila e Pilha**. 2014. Disponível em: <http://slideplayer.com.br/slide/358853>. Acesso em: 11/05/2018.

MANZANO, J. A. G.; COSTA JR., R. **Programação de Computadores com Java**. São Paulo: Érica, 2014. 127p.

MOZILLA. O que é JavaScript? 2016. **Portal MDN** web docs Mozilla. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/About_JavaScript>. Acesso em: 13/05/2018.

ORACLE. Controlling Access to Members of a Class. **Portal Oracle** – Java Documentation, 2018a. Disponível em: <https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>. Acesso em: 10/05/2018.

ORACLE. Duke, the Java Mascot. Oracle. **Portal Oracle**, 2018b. Disponível em: <https://www.oracle.com/java/duke.html>. Acesso em: 13/05/2018.

PAGLIARES, R. **Introdução à coleções em Java**. Canal Rodrigo Pagliares, YouTube, publicado em 13 de set de 2012. Disponível em: <https://www.youtube.com/watch?v=m6JNiSY8rrE>. Acesso em: 14/05/2018.

PALMEIRA, T. V. V. Entendendo e conhecendo as versões do Java. Portal **DevMedia**, 2012. Disponível em: <https://www.devmedia.com.br/entendendo-e-conhecendo-as-versoes-do-java/25210>. Acesso em: 15/08/2018.

PALMEIRA, T. V. Trabalhando com a Interface Set no Java. **Portal Linha de Código**, 2018. Disponível em: <http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx>. Acesso em: 15/08/2018.

REGI JUNIOR, S. **Utilização dos métodos de orientação a objetos na gestão dos recursos humanos envolvidos em projetos nas empresas**. 2009. Disponível em: <http://www.administradores.com.br/artigos/carreira/utilizacao-do-metodo-de-orientacao-a-objetos-na-gestao-dos-recursos-humanos-envolvidos-em-projetos-nas-empresas/37010/>. Acesso em: 15/05/2018.

RICARTE, I. L. M. **Bytecodes**. Universidade de Campinas. Departamento de Engenharia de Computação e Automação Industrial (DCA). Faculdade de Engenharia Elétrica e de Computação. Campinas: DCA/FEEC/UNICAMP, 2017. Disponível em: <http://www.dca.fee.unicamp.br/cursos/PooJava/javaenv/bytecode.html>. Acesso em: 14/05/2018.

ZUKOWSKI, J. **Java Collections Framework**. ibm.com/developerWorks, 2001. Disponível em: <http://pages.di.unipi.it/corradini/Didattica/PR2-B-14/Java%20Collections%20Framework.pdf>. Acesso em: 15/08/2018.