



Conceitos de Programação Orientada a Objetos em Java


A declaração e uso de métodos/funções no Java exigem o cumprimento de algumas regras que se assemelham a boa parte das linguagens de programação. No entanto, ao contrário do JavaScript, por exemplo, a declaração de um método no Java não exige o uso de nenhuma palavra reservada, exige, porém, que seja informado o tipo de retorno que ela terá. Ou seja, se um método tem como resultado uma String, por exemplo, este tipo tem que ser informado antes do nome que será dado ao método, como mostrado no código abaixo.

```
String uneDuasStrings(String a, String b) {  
    return a + b;  
}
```

Neste método, os parâmetros a e b, ambos Strings, são concatenados (unidos) e retornados (o que é indicado pela palavra reservada *return*). No entanto, existem métodos que não possuem retorno, eles apenas realizam algumas operações e finalizam por si só. Nestes casos, é usada a palavra reservada *void*. Por exemplo:

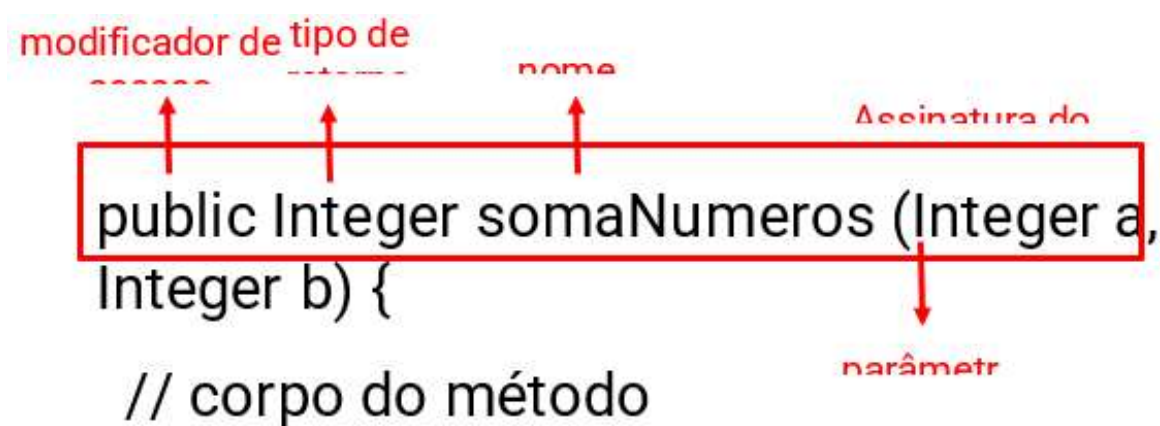
```
void exibeMensagemNoConsole() {  
    System.out.println("mensagem");  
}
```

Além do tipo de retorno e o nome, o método também deve indicar quais são os parâmetros (e seus respectivos tipos) que devem ser passados para que ele

funcione. No primeiro exemplo, a função *uneDuasStrings()* têm dois parâmetros listados como *a* e *b* (a escolha do nome dos parâmetros também é arbitrária).  No segundo exemplo, a função *exibeMensagemNoConsole()* não tem nenhum parâmetro, logo, os parênteses ficaram vazios. Não há um limite para o número de parâmetros que um método pode ter, mas, uma grande quantidade pode indicar uma falha na arquitetura do código.


A declaração de um método também conta com o modificador de acesso no início que sinaliza quem poderá ter acesso a ele. No final, um método tem a seguinte estrutura:

Figura 1 - Estrutura de um método no Java.



Toda a parte do método que está fora do seu corpo (antes das chaves) é chamado de **assinatura do método**. São as informações que indicam para o usuário como usá-la, por isso possuem este nome. Muitos também se referem como sendo o contrato para uso do método. Esta estrutura é primordial no uso de **interfaces** na orientação a objetos.

Depois de declarada, uma função precisa ser chamada para que possa ser executada. Esse processo é chamado de **invocação** de um método. Para isso, basta usar o nome do método acompanhado dos parênteses. Se o método exigir

parâmetros, eles devem ser inseridos dentro dos parênteses, sem a necessidade de indicar o tipo. No exemplo abaixo, a função *soma()* recebe dois parâmetros por **referência**, ou seja, um ponteiro para a referência em memória das informações é enviado de tal modo que a função consegue acessar e manipular esses valores. 

```
public Integer soma(Integer a, Integer b) {  
    return a + b;  
}  
  
soma(1, 3) // 4
```

Outro aspecto interessante sobre os métodos no Java é que eles podem ser sobrescritos. Na prática, isso acontece quando há herança entre classes, ou seja, uma classe pai possui métodos que podem ter seu comportamento alterado nas classes filhas. Como todos os objetos no Java são extensões da classe *Object*, já existem alguns métodos que podem ser sobrescritos por qualquer classe. Esse processo é chamado de **sobrecarga** e é marcado no Java com a notação *@Override*. Um exemplo é o método *equals()* que pode – e em muitas situações deve – ser sobrescrito para descrever como dois objetos se diferenciam.

```
@Override  
public boolean equals(Object obj) {  
    return this.nome == ((Aluno) obj).getNome();  
}
```

Por fim, métodos podem invocar a si mesmos. Esse processo é chamado **recursividade**. No exemplo abaixo, um método invoca a si mesmo até chegar no resultado.



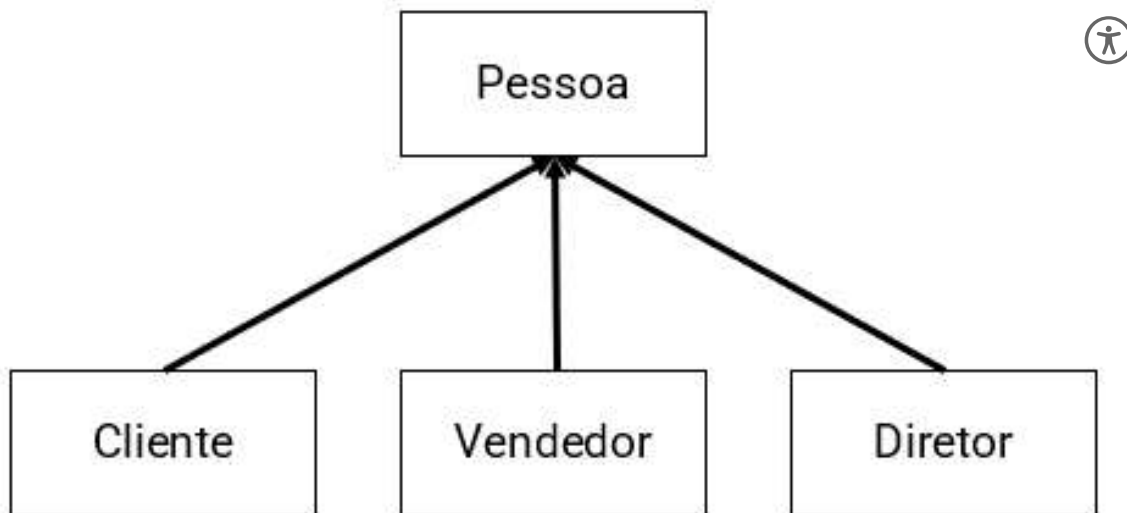
```
public void imprimirSequencia(Integer x) {  
    if (x == 0) return;  
    System.out.println(x);  
    imprimirSequencia(x - 1);  
}
```

Herança

A **herança** é um mecanismo disponível nas linguagens de programação que se apoiam no paradigma de objetos que possibilita que características comuns a diversas classes sejam reunidas em uma classe base. A partir desta base, as demais classes podem ser especificadas de modo que cada classe derivada (ou subclasse) apresente características e comportamentos herdados da classe base e outras que foram definidas exclusivamente para elas. Na prática, isso significa reunir todas as características comuns entre várias entidades e a partir disso criar uma base em que todas elas se baseiam. Em resumo, as características e comportamentos de uma classe (base) são herdadas a outras classes (subclasses).

Um exemplo prático clássico é uma possível entidade Pessoa em sistemas que exigem cadastro. Geralmente a classe que representa uma pessoa física tem características comuns, tais como: nome, idade, endereço... No entanto, é possível abstrair desta entidade características mais específicas de acordo com o papel dela em uma empresa, por exemplo: vendedor, cliente, diretor... e assim por diante. Cada um destes papéis podem ter características e comportamentos diferentes, no entanto, todos eles parte do princípio de que é uma pessoa atuando, logo, apesar das particularidades, todos possuem nome, idade, endereço e todas as demais características da classe base.

Figura 2 - Herança da classe Pessoa para as subclasses.



Herança é sempre utilizada em Java, mesmo que não explicitamente. Quando uma classe é criada e não há nenhuma referência à sua classe base, implicitamente a classe criada é derivada diretamente da classe *Object*. É por esse motivo que todos os objetos podem invocar os métodos da classe *Object*, tais como *equals()* e *toString()*.

Além disso, outro mecanismo importante para a orientação a objetos são as **interfaces**. A sua função é fazer com que todas as classes associadas a elas sejam “obrigadas” a cumprir um "acordo". Este acordo será possuir as características e comportamentos definidos por ela. Se ela diz que todos devem ter um método *pagar()*, por exemplo, todas as classes associadas a esta interface terão que, obrigatoriamente, implementar este método. O mesmo vale para atributos.

Um exemplo bem claro é a interface *List* do Java. Esta interface define uma série de métodos, tais como:

- `size();`
- `isEmpty();`
- `contains();`
- `add();`

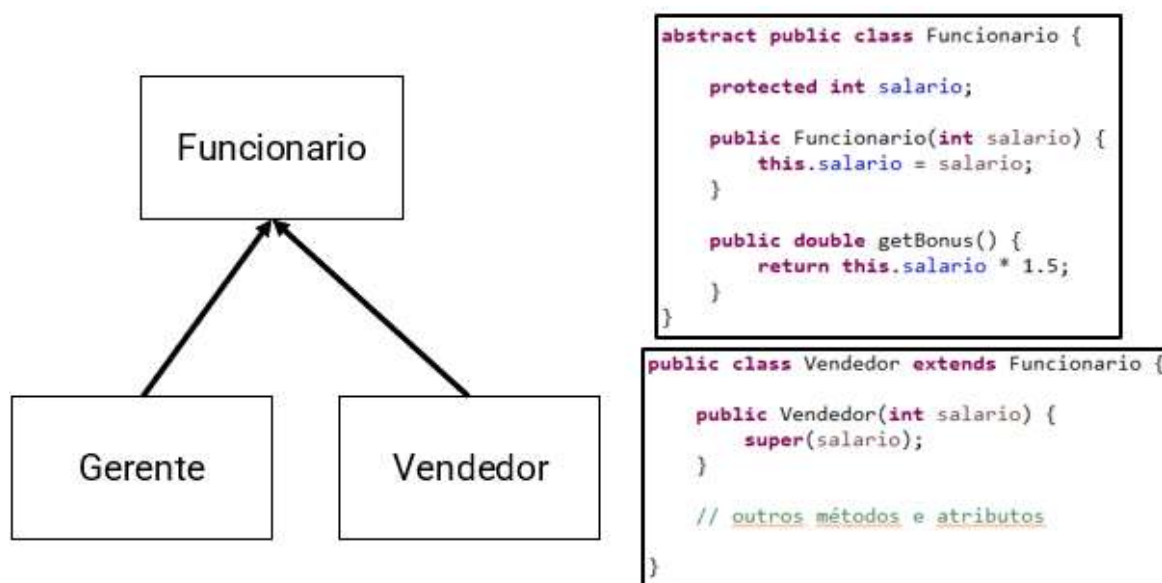
- `remove()`.



Todos estes métodos não possuem implementação dentro da interface, são somente a assinatura dele. As classes que implementam essa interface são as responsáveis por implementar esses métodos, como é o caso das classes *ArrayList*, *Vector* e *LinkedList*.

Apesar da versatilidade das classes através da herança e das interfaces, existem situações em que fará sentido o uso de **classes abstratas** e **anônimas**. As abstratas são classes que estruturam um tipo mas que não podem ser instanciadas. É quase como se fosse uma interface, a diferença é que elas podem implementar comportamentos e características, mas nunca serão usadas diretamente, somente as classes que a herdarem poderão usá-la. É como no exemplo a seguir:

Figura 3 - Exemplo de implementação de uma classe abstrata.



Neste caso, a classe *Funcionario* nunca poderá ser instanciada, somente aquelas que a estendem, no caso, *Gerente* ou *Vendedor*, como mostra o exemplo abaixo:

```
public class Main {  
    public static void main(String[] args) {  
        Funcionario vendedor1 = new Vendedor(1000);  
        double bonus = vendedor1.getBonus();  
        System.out.println(bonus);  
    }  
}
```



Já as classes anônimas são classes internas a outras classes e que não possuem um nome. Por exemplo, ao levar em consideração uma interface chamada `ListalInventada`, é possível criar uma classe interna que implementa esta interface e então já utilizá-la em seguida.

```
import java.util.List;  
  
public interface ListalInventada {  
    Integer quantidadeItens();  
    void setItens(List<Integer> itens);  
}
```



```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> numeros = new ArrayList<Integer>();
        numeros.add(1);
        numeros.add(2);
        numeros.add(3);

        ListaInventada lista = new ListaInventada() {

            private List<Integer> itens;

            @Override
            public Integer quantidadeItens() {
                return this.itens.size();
            }

            @Override
            public void setItens(List<Integer> itens) {
                this.itens = itens;
            }
        };

        lista.setItens(numeros);
        Integer quantidade = lista.quantidadeItens();

        System.out.println(quantidade);
    }
}
```

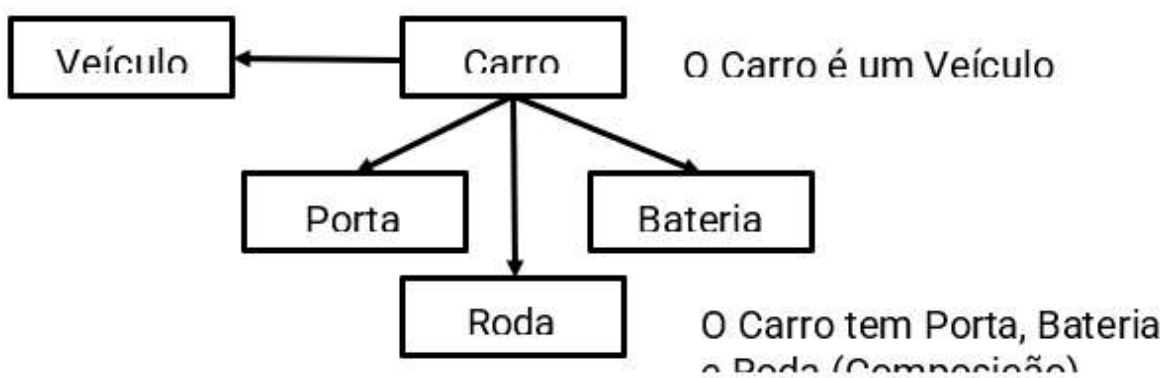
Por fim, é importante lembrar que a notação *@Override* representa a sobreposição de um método. No caso das interfaces sendo implementadas, isso é mais simples, pois as interfaces não possuem implementação (a partir do Java 8 é possível criar

uma implementação padrão que pode ser sobrescrita). Mas quando uma classe estende a outra, ela tem a liberdade de sobrescrever um método, ou seja, usar a mesma assinatura para dar um resultado diferente. É o que acontece com os métodos iguais da classe *Object*. Qualquer classe que é declarada no Java pode sobrescrever este método e definir a forma como dois objetos são comparados.

Composição

Além da herança, a programação orientada a objetos possui outro mecanismo importante para associar e reutilizar classes diferentes. Um dos mais importantes, e que bate de frente com a herança, é a **composição**. Como o próprio nome já indica, a ideia de composição é compor uma classe usando outras classes, ou seja, objetos dentro de objetos. A composição é tida como um relacionamento do tipo “tem um(a)”, ou seja, uma classe x tem uma classe y; diferente do tipo de relacionamento de uma herança, onde o relacionamento é do tipo “é um(a)”, ou seja, uma classe x é um tipo da classe y. A imagem abaixo é um exemplo claro de uma situação onde herança e composição estão sendo utilizadas ao mesmo tempo.

Figura 4 - Exemplo de classes relacionadas por herança e por composição. Autoral.

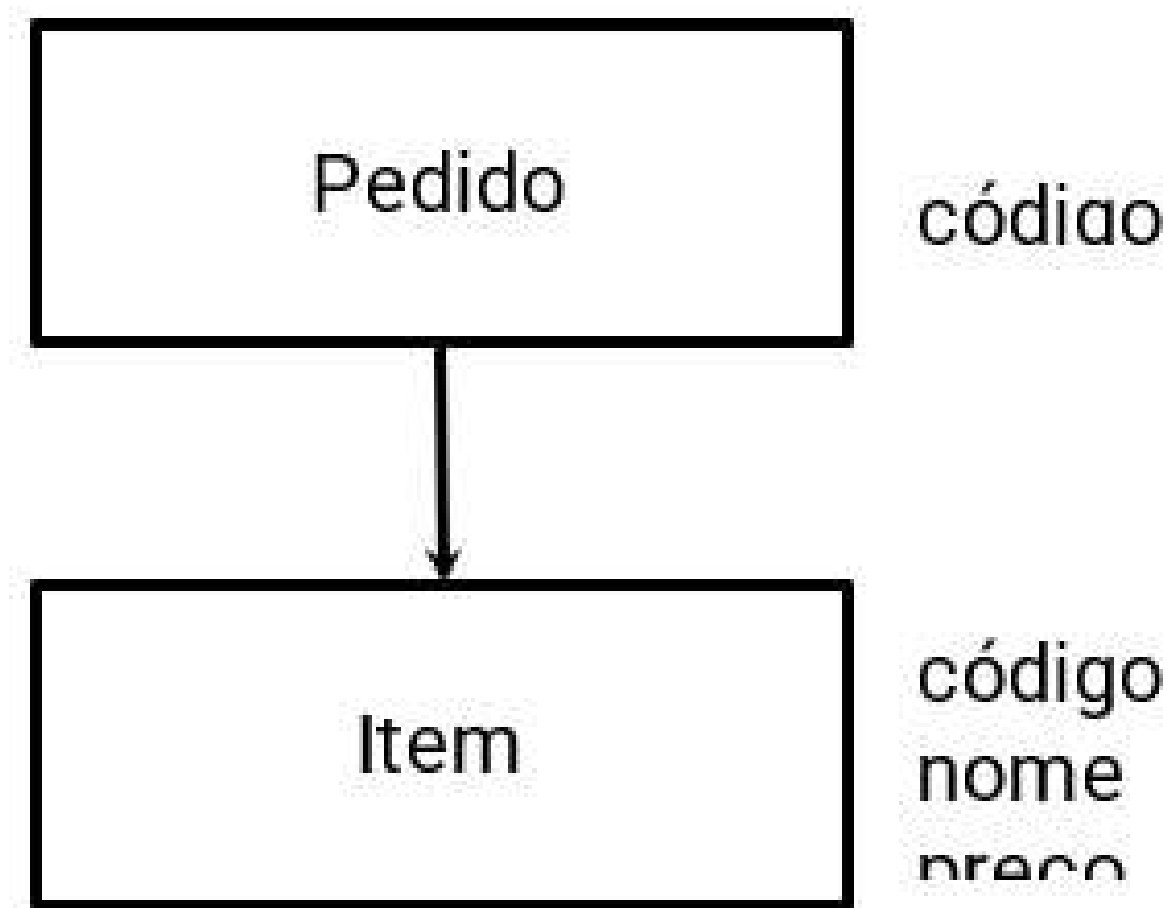


Um outro exemplo prático simples é uma suposta classe Pedido que contém itens e estes itens são compostos por uma classe chamada Item que especifica o nome, preço, código de barras e quaisquer outras informações relevantes. Ou seja, aqui a classe Pedido se livra da responsabilidade de saber quais os nomes dos itens, seus

preços e coisas do gênero, ela simplesmente usa a classe Item que tem essa responsabilidade. Esta representação fica mais clara no esquema abaixo:



Figura 5 - Representação da classe Pedido. Autoral.



Em termos de código, essa relação poderia ser implementada da seguinte forma (usando classes internas somente para evitar um código de exemplo muito fragmentado):

```
import java.util.ArrayList;
```

```
import java.util.List;
```

```
public class Main {
```

```
    static class Pedido {
```

```
        private String identificador;
```

```
        private List<Item> itens;
```





```
public Pedido(String identificador) {  
    this.identificador = identificador;  
    this.itens = new ArrayList<Item>();  
}
```

```
public void addItem(Item item) {  
    this.itens.add(item);  
}
```

```
@Override  
public String toString() {  
    String itens = "";  
    for(Item item : this.itens) {  
        itens += item.nome;  
    }  
  
    return itens;  
}  
}
```

```
static class Item {  
    private String codigo;  
    private String nome;  
    private double preco;  
  
    public Item(String codigo, String nome, double preco) {  
        this.codigo = codigo;  
        this.nome = nome;  
        this.preco = preco;  
    }  
}
```



```
}  
}  
  
public static void main(String[] args) {  
    Pedido pedido = new Pedido("1231");  
    pedido.addItem(new Item("Pão", "Pão", 2.90));  
    System.out.println(pedido.toString());  
}  
}
```

Da mesma forma que neste exemplo a classe Pedido tem uma lista de classe do tipo Item, este relacionamento poderia ir muito além e envolver uma quantidade maior de classes, o que acaba acontecendo nas aplicações do mundo real.

Em resumo, uma composição:

- Contém uma classe e delega o trabalho para o objeto desta classe;
- Uma instância da classe existente é usada como componente da outra classe;
- Relacionamento do tipo “tem”.

Uma das discussões que acontecem no planejamento de uma aplicação é quando usar a herança e quando usar a composição. As boas práticas dizem que o uso da composição é favorável por alguns motivos, sendo eles:

- Os objetos que foram instanciados e estão contidos na classe que os instanciou são acessados somente através de sua interface;
- A composição pode ser definida dinamicamente em tempo de execução pela obtenção de referência de objetos a objetos de do mesmo tipo;

- A composição apresenta uma menor dependência de implementações;



- Na composição temos cada classe focada em apenas uma tarefa;
- Na composição temos um bom encapsulamento visto que os detalhes internos dos objetos instanciados não são visíveis.

Atividade Extra

- Vídeo: Java- Methods
- Vídeo: Java Recursion

Referência Bibliográfica

BARNES, D. J. KOLLING, M. **Programação orientada a objetos com java: uma introdução prática usando o bluej**. 4.ed. Pearson: 2009.

FELIX, R. (Org.). **Programação orientada a objetos**. Pearson: 2017.

MEDEIROS, L. F. de. **Banco de dados: princípios e prática**. Intersaberes: 2013;

ORACLE. Java Documentation, 2021. **Documentação oficial da plataforma Java**. Disponível em: <<https://docs.oracle.com/en/java/>>

Ir para exercício

