Java e Orientação a Objetos

Introdução

Java desde sua criação foi feita para ser uma linguagem puramente orientada a objetos por isso que dizemos que tudo em java pode ser uma classe e por sua vez um objeto. Mas o que é a orientação a objetos afinal?

Orientação a objetos é um paradigma de programação que tenta imitar o comportamento humano em relação ao mundo físico que o cerca. Essa explicação parece complexa, mas tente imaginar o seu dia a dia, estamos o tempo todo nos orientado a objetos que estão à nossa volta, por exemplo: digamos que você esteja com sede, para matar essa sede é necessário beber água (nosso primeiro objeto), porém para beber a água nós precisamos de um copo (nosso segundo objeto) e por último precisamos da torneira que fornece essa água (nosso terceiro objeto).

Olha só que incrível, apenas para beber uma água tivemos a interação com 3 objetos totalmente diferentes entre eles, mas que juntos atingem o mesmo objetivo.

Essa é a ideia dentro do paradigma de orientação a objetos, é literalmente trabalhar com o código e criar “objetos” que façam as coisas acontecerem.

Classes

Vimos na introdução que a ideia da programação orientada a objetos é justamente criar objetos que interagem entre si e consigam resolver os problemas para os quais eles foram criados. Mas então onde entra a “classe” nisso tudo?

Enquanto os objetos são coisas específicas com características únicas as classes é a forma que podemos classificar esses tipos de objetos para que assim, no java, eles possam ser criados.

Aproveitando o exemplo da introdução onde falamos do copo para armazenar a água, podemos imaginar como seria a classificação dos objetos “Copo” e fazemos isso analisando suas características. Vamos pensar. Quais são as características que todo copo tem? Largura, profundidade, cor, material etc. Essas são algumas das características que podemos pensar.

Agora tente imaginar um copo usando esses atributos, nesse momento podem ter a certeza de que cada um acaba imaginando um copo diferente, um pode imaginar um copo fundo e outro razo, pode imaginar um copo feito de vidro e outro de plástico e assim por diante, e o mais importante todos são copos e sabemos disse como conseguimos classificar eles utilizando uma Classe com os atributos largura, profundidade, cor e material, mas agora os objetos de classe podem e serão diferentes entre si.

2.1 - Como Criar uma Classe

Para criar essas classe no java podemos abrir nosso projeto e dentro dele localizar o pacote principal do projeto, onde a classe que carrega o método Main existe, e ali podemos criar mais uma classe.

Na maioria das IDEs você pode criar essa classe apenas clicando com o botão direito e selecionando “criar classe java” ou “new file”.

Como exemplo, vou criar uma classe chamada Contador, essa classe deverá criar objetos que são capazes de somar 2 número que são passados para ele.

Vai ficar mais ou menos assim:

```
package com.br.zup;

public class Contador {
}
```

Já vemos logo de cara que temos algumas regras para criar uma classe. A primeira é colocar a para “class” logo depois do “public” isso serve para indicar ao java que esse código se trata de uma classe.

A segunda regra é: toda classe tem a primeira letra do nome em maiúsculo, se tiver mais de uma para então é primeira letra de cada palavra em MAIÚSCULO. Isso é chamado de **PascalCase**.

2.2 - Adicionando Atributos

Para essa classe “Contador” precisamos adicionar as características dele, nesse caso como nosso objetivo é apenas somar 2 números então podemos dizer

que suas características são “primeira parcela” e “segunda parcela” que são os nomes dados em uma operação de soma.

```
package com.br.zup;

public class Contador {
   int primeiraParcela;
   int segundaParcela;

}
```

Aí estão nossos atributos, podemos perceber que com eles colocamos apenas em maiúsculo a segunda palavra do nome, chamamos isso de **camelCase**.

2.3 - Criando Método

Agora vamos criar o método que faz a soma da primeira parcela e da segunda parcela e retorna para o usuário o valor total.

```
package com.br.zup;

public class Contador {
   int primeiraParcela;
   int segundaParcela;

   public int somarAsParcelas(){
       int total = primeiraParcela + segundaParcela;
       return total;
   }
}
```

Os métodos das classes são os comportamentos que os objetos criados tem. Ou seja, são através dos métodos que os objetos vão interagir com o código e possibilitar a execução de tarefas. No nosso caso o método somar Parcelas vai possibilitar ao objeto contator somar 2 valores.

2.4 - Instanciando um Objeto

Dentro da classe principal do projeto, a que carrega o método main, podemos instanciar nosso objeto e assim utilizar sua capacidade de somar 2 valores.

```
package com.br.zup;

public class Main {

   public static void main(String[] args) {

       Contador contador = new Contador();
   }
}
```

Podemos notar que antes de instanciar o objeto contador foi necessário criar uma variável que recebe a classe Contador como tipo, e logo depois do sinal de igual instânciamos um objeto Contador usando a palavra “new” e o nome da classe Contador seguido de abre e fecha parênteses.

2.5 - Preenchendo os Atributos do Objeto

Sempre que criamos um objeto a partir de uma classe os atributos são iniciados, sendo assim é necessário preencher esses atributos com um valor para inicializá-los.

```
package com.br.zup;

public class Main {

   public static void main(String[] args) {

       Contador contador = new Contador();
       contador.primeiraParcela = 13;
       contador.segundaParcela = 2;

   }
}
```

No exemplo acima inserimos dois valores, um em cada atributo da classe. Em tempo de execução do código fizemos isso de maneira separada da criação do

objeto. Então em primeiro momento tínhamos um objeto sem nenhum atributo preenchido, agora temos o objeto com ambos os atributos preenchidos.

2.6 - Chamando o Método do Objeto

Assim como podemos “chamar” o atributo do objeto através do ponto, podemos também “chamar” os métodos que esse objeto tem.

```
public class Main {

    public static void main(String[] args) {
        Contador contador = new Contador();
        contador.primeiraParcela = 13;
        contador.segundaParcela = 2;

        int total = contador.somarAsParcelas();
        System.out.println(total);

    }
}
```

Como podemos ver no exemplo assim, podemos chamar o método de **somarAsParcelas** usando o ponto logo após o nome do nosso objeto. Como esse método retorna um número inteiro, podemos armazenar esse número em uma variável, que no caso chamamos de total, e depois usar ela para imprimir na tela para o usuário.

2.7 - Conclusão

Com as classes podemos criar estruturas de dados que simulam objetos em nosso código, é através das classes que podemos dar as características em forma de atributos e comportamentos em forma de métodos para nossos objetos.

É sempre bom lembrar que cada classe deve ter um propósito para existir no nosso código, não existe um limite de classes para serem feitas, mas existe a questão de coerência na existência delas.

3 - Encapsulamento

Quando trabalhamos com programação orientada a objetos, precisamos nos preocupar com outros conceitos que estão ligados diretamente à orientação a

objetos. Um desses conceitos é o encapsulamento, o princípio dele é trazer a preocupação com o acesso aos atributos e métodos dentro de uma classe garantindo que os atributos não sejam alterados durante a execução ou que um método restrito da classe não seja executado.

Para bloquear esses acessos devemos usar os modificadores de acesso.

3.1 - Modificadores de Acesso

Os modificadores de acesso são responsáveis, como dito anteriormente, por controlarem os acessos aos atributos, métodos ou até mesmo às classes.

Modificador	Detalhes
public	O modificador ***public*** permite que todas as outras classes dentro ou fora do mesmo pacote que a classe dona do atributo ou medito tenha acesso a eles.
private	O modificador ***private*** impede que qualquer classe dentro ou fora da mesma classe dona do atributo ou método tenha acesso a eles.
protected	O modificador ***protected*** permite que **apenas** as classes **dentro do mesmo pacote** que a classe dona do atributo/método tenha acesso a eles.
Padrão (quando não declarado o modificador)	O comportamento de acesso **quando não declaramos um modificador** é parecido com o do protected, apenas as classes e interfaces dentro do mesmo pacote podem acessar o atributo ou método.

Exemplo:

```
package com.br.zup;

public class Contador {
   private int primeiraParcela;
   private int segundaParcela;

   public int somarAsParcelas(){
       int total = primeiraParcela + segundaParcela;
```

```
        return total;
    }
}
```

Declarando os atributos da classe contador como privados, não podemos mais realizar acessar livremente os atributos e preenchê-los como antes, utilizando o nome do objeto e os atributos.

```
// NÃO FUNCIONA MAIS
Contador contador = new Contador();
        contador.primeiraParcela = 13;
        contador.segundaParcela = 2;
```

Agora para que possamos preencher os atributos desse objeto devemos contar com o método construtor, já usamos ele naturalmente quando instânciamos a classe, ele está ali presente em todos os exemplos usados até agora.

Quando usamos "`new Contador();`" estamos justamente chamando o método construtor que literalmente constrói a instância da nossa classe.

OBS: a palavra "`new`" sempre vai representar uma nova instância de um objeto.

3.2 - Método Construtor

Agora com os modificadores de acesso ao atributo da nossa classe Contador, devemos criar um novo construtor para preencher esses atributos.

Por padrão, quando não criamos nenhum método construtor o java implementa um construtor "vazio" que apenas instância o objeto, mas não preenche os atributos. Nesse caso criaremos o nosso que preencherá os atributos do objeto ao mesmo tempo que o cria.

Dentro da classe vamos criar então um método que carrega o modificador de acesso ***public*** e que carrega **exatamente o mesmo nome da classe**.

Também vamos adicionar 2 parâmetros nesse método, são pelos parâmetros que vamos receber os valores e usá-los no método para preencher nossos atributos.

```
public class Contador {
    private int primeiraParcela;
    private int segundaParcela;

    public Contador(int primeiroNumero, int segundoNumero){
        this.primeiraParcela = primeiroNumero;
        this.segundaParcela = segundoNumero;
    }
}
```

Dessa maneira nossa classe principal com o método main ficará dessa forma.

```
public class Main {

    public static void main(String[] args) {
        Contador contador = new Contador(13, 2);

        int total = contador.somarAsParcelas();
        System.out.println(total);

    }
}
```

3.3 - Sobrecarga de Construtores

Em alguns casos temos a necessidade de ter mais de uma forma de construir um objeto de uma classe. Nesses casos contamos com a sobrecarga de construtores que permite realizar diferentes tipos de construções de instâncias de uma classe.

```
public class Contador {
    private int primeiraParcela;
    private int segundaParcela;

    public Contador(int primeiroNumero, int segundoNumero){
```

```
        this.primeiraParcela = primeiroNumero;
        this.segundaParcela = segundoNumero;
    }

    public Contador(int primeiraParcela){
        this.primeiraParcela = primeiraParcela;
        this.segundaParcela = 0;

    }

}
```

No exemplo acima temos 2 construtores, um que recebe 2 parâmetros no método e outro que recebe apenas 1 parâmetro. O java vai saber qual método construtor utilizar pela assinatura do método, ou seja, pela quantidade de parâmetros que ele receberá.

Exemplo:

```
public class Main {

    public static void main(String[] args) {
        Contador contadorUm = new Contador(13, 2);
        Contador contadorDois = new Contador(30);
    }
}
```

3.4 - Métodos Gets e Setters

Outra forma de dar acesso a atributos privados é através dos métodos Get e Set de cada atributo. Um método get é criado para retornar o valor do atributos enquanto o método set é feito para alterar o valor de um atributo.

- Get = Pegar
- Set = Inserir

Esses métodos são totalmente opcionais e devem ser criados apenas se for necessário o acesso a esse atributo, isso faz sentido quando pensamos que alguns atributos de controle de classes não devem ser alterados, apenas visualizados (get)

ou apenas alterados e não visualizados (set). Os métodos set e get também podem carregar alguma regra de negócio específica antes de alterar o valor no caso do set, por exemplo, assim garantimos que o atributo será preenchido apenas com valores corretos.

```
public class Contador {
    private int primeiraParcela;
    private int segundaParcela;

    public int getPrimeiraParcela() {
        return primeiraParcela;
    }

    public void setPrimeiraParcela(int primeiraParcela) {
        this.primeiraParcela = primeiraParcela;
    }

    public int getSegundaParcela() {
        return segundaParcela;
    }

    public void setSegundaParcela(int segundaParcela) {
        this.segundaParcela = segundaParcela;
    }
}
```

É muito importante notar o padrão dos nomes dos métodos Get e Set pois esse padrão é usado para funcionamento de alguns framework como Spring que procura esses métodos para criar ou receber um Json, mas isso falaremos mais adiante.

4 - Herança

A herança é um dos principais pilares de uma linguagem orientada a objetos. Através dela, é possível realizar reaproveitamento de códigos através da criação de uma classe base (Também conhecida como superclasse), que permite que diversas outras classes (chamadas de subclasses ou classes derivadas) recebam características em comum de sua classe mãe. Na prática, este mecanismo permite

que você possa através de sua classe base, especificar em suas subclasses suas características específicas, visto que herdarão suas características em comum de sua superclasse.

4.1 - Utilizando Herança

Suponhamos que nós fomos contratados para desenvolver um sistema de gestão para a escola X. Em uma das etapas, nós percebemos que precisaríamos criar duas classes: uma classe chamada técnicos administrativos e uma classe chamada professores. Por mais que ambas possuam características e permissões diferentes, ambas possuem nome, telefone e cpf, por exemplo.

Neste caso, ao invés de criar duas classes independentes com os mesmos atributos, podemos criar uma classe "mãe" chamada funcionário, na qual professores e técnicos administrativos irão herdar todas as características que eles possuem em comum. Isso não significa que eles terão as mesmas caraterísticas, afinal, alunos e professores ainda são classes independentes, porém, ambos terão nome, cpf e telefone que herdaram de sua classe base.

Primeiro, vamos observar o conteúdo da classe funcionário, e posteriormente, vamos criar uma relação de herança entre a classe Funcionário e a classe professores, alegando que um professor **é um** funcionário.

```
public class Funcionario {
    private String nome;
    private String telefone;
    private String cpf;

    public Funcionario(String nome, String telefone, String cpf) {
        this.nome = nome;
        this.telefone = telefone;
        this.cpf = cpf;
    }

    public String getNome() {
        return nome;
    }

    public String getTelefone() {
        return telefone;
    }

    public String getCpf() {
        return cpf;
    }
```

```
}
```

No código acima, podemos identificar a criação da classe “Funcionario” com 3 atributos privados: nome, telefone e CPF com seus respectivos getters. No código abaixo, vamos estabelecer uma relação de herança entre a classe ***Funcionario*** e suas subclasses: ***Professores*** e ***TecnicosAdministrativos*****.**

```
public class Professores extends Funcionarios {

// Conteúdo da subclasse Professores

}

public class TecnicosAdministrativos extends Funcionarios
{

// Conteúdo da subclasse Técnicos administrativos

}
```

Neste caso, ainda que você não tenha declarado os atributos dentro das classes ***Professores*** e ***TecnicosAdministrativos,*** a mesma possui todo o conteúdo herdado de sua classe mãe através do comando “**extends**”. Mas como podemos acessá-los? Para tal, é necessário dentro de suas subclasses fazer referência aos atributos e métodos de sua superclasse.

4.2 - This e Super

Com certeza, em algum momento deste material, você encontrou o comando “This” em algum método criado anteriormente, naturalmente, este comando é fundamental na criação dos “setter” em nosso código. Afinal, a responsabilidade deste comando é fazer referência a algum atributo encapsulado dentro de sua própria classe, como por exemplo, se quisermos criar os setters em nossa classe funcionários.

```
public class Funcionario {
   private String nome;
   private String telefone;
   private String cpf;

   public Funcionario(String nome, String telefone, String cpf) {
       this.nome = nome;
       this.telefone = telefone;
       this.cpf = cpf;
   }

   public String getNome() {
       return nome;
   }

   public String getTelefone() {
       return telefone;
   }

   public String getCpf() {
       return cpf;
   }
}

public void setNome(String nome){
      this.nome = nome;
}

public void setTelefone(String telefone){
      this.telefone = telefone;
}

public void setCpf(String cpf){
      this.cpf = cpf;
}
```

Repare que dentro do **setNome**, nós recebemos uma string como argumento e inserimos essa string recebida dentro do `this.nome`. Neste caso, como o comando **this** foi acionado, nós estamos incrementando o argumento recebido diretamente no atributo declarado em cima da classe, é como se estivéssemos fazendo uma ligação direta ao `private String nome;`, conectando o conteúdo do método ao dado encapsulado anteriormente. Uma informação interessante, é que o comando this é opcional quando, no método, o nome dos parâmetros são diferentes dos atributos.

Já o comando Super, ao invés de fazer referência a um atributo dentro da própria classe, ele faz referência direta a atributos e métodos presentes dentro de sua superclasse, na qual você estabeleceu uma relação de herança. Em termos técnicos, você estará invocando o construtor de sua superclasse (Funcionários), fazendo com que professores e técnicos administrativos tenham acesso aos dados então encapsulados de sua classe base.

```
public class Professor extends Funcionario {
  public Professor(String nome, String telefone,
                   String cpf){
      super(nome, telefone, cpf);
  }

}

public class TecnicosAdministrativos extends Funcionario {
  public TecnicosAdministrativos(String nome, String
                                 telefone, String cpf){
      super(nome, telefone, cpf);
  }

}
```

No exemplo acima, podemos ver uma aplicação clássica do comando **super**. Quando criamos o método construtor de nossa classe, nós invocamos também através do comando super as características herdadas da classe mãe, sendo elas: nome, telefone e CPF. Com isso, conseguimos através da herança aproveitar os atributos criados dentro da classe base em nossas subclasses, diminuindo o tempo de criação e aumentando a qualidade do código.

Em resumo, o ***this*** possui a responsabilidade de criar uma conexão entre o método e os atributos encapsulados dentro de sua própria classe, enquanto o ***super***, é utilizado sempre que desejamos acessar/manipular atributos e métodos estruturados na classe mãe de nossa subclasse em questão.

5 - Polimorfismo

A grosso modo, polimorfismo significa "muitas formas". Na prática, o termo é utilizado quando temos duas ou mais classes derivadas da mesma superclasse compartilhando métodos de mesma assinatura, porém, com comportamentos diferentes.

Para compreendermos melhor o polimorfismo na prática, vamos trabalhar com um exemplo clássico: através de formas geométricas. Inicialmente, vamos criar uma classe genérica chamada "***FormasGeometricas***" e vamos estabelecer uma relação de herança entre esta classe com suas duas subclasses: ***Circulo*** e ***Quadrado***.

```
public class FormasGeometricas {

}

public class Circulo extends FormasGeometricas {

}

public class Quadrado extends FormasGeometricas {

}
```

Após criar as classes, vamos criar um método que será responsável por calcular a área de cada uma das formas geométricas. Porém, temos um grande problema:

- Para calcular a área do quadrado, nós necessitamos de um lado, afinal, um quadrado nada mais é do que uma forma de quatro lados iguais.
- Já para calcular a área do círculo, nós precisamos do raio, que é a distância do centro até a borda do círculo.

Neste caso, precisamos de um método comum para ambas as classes que deve respeitar as suas características.

```
public class FormasGeometricas {
   public double calcularArea(){
```

```
        return 0;
    }
}
```

No código acima, criamos um método na classe base que por enquanto vai retornar 0, afinal, vamos reescrevê-lo nas subclasses com a mesma assinatura mas com comportamentos diferentes, respeitando as especificidades de cada classe. Este processo é conhecido como “polimorfismo dinâmico”, onde uma subclasse redefine um método presente na superclasse.

```
public class Circulo extends FormasGeometricas {
    double raio;

    public Circulo(double raio){
        this.raio = raio;
    }

    @Override
    public double calcularArea() {
        double area = 0;
        area = 3.14 * raio * raio;
        return area;
    }
}
```

No código acima, reescrevemos o método criado na superclasse dentro da subclasse **Circulo.** O método possui a mesma assinatura, porém, o conteúdo é diferente, retornando o cálculo da área do círculo (3.14 * raio * raio).

```
public class Quadrado extends FormasGeometricas {
    double lado;

    public Quadrado (double lado){
        this.lado = lado;
    }

    @Override
    public double calcularArea() {
        double area = 0;
```

```
        area = lado * lado;
        return area;
    }
}
```

No código acima, podemos observar também a sobreposição do método ***CalcularArea,*** onde o método possui a mesma assinatura de sua classe base, porém, retornando o cálculo da área do quadrado (lado * lado). Como podemos perceber em ambas as classes, sempre que um método é sobreposto, a palavra `@Override` aparece por cima do método, representando a sobreposição.

6 - Classes Abstratas

Classes abstratas são classes genéricas que possuem um nível tão alto de abstração que não faz mais sentido criar objetos a partir dela. Uma grande característica das classes abstratas é que elas não podem ser instanciadas, apenas herdadas, servindo apenas como molde para classes concretas que serão geradas a partir delas, como podemos observar no exemplo abaixo:

```
public abstract class Animal{
    // Conteúdo da classe
}

public class Baleia extends Animal{
    // Conteúdo da classe
}

public class Cachorro extends Animal{
    // Conteúdo da classe
}

public class Papagaio extends Animal{
    // Conteúdo da classe
}
```

No exemplo acima, podemos observar a criação de uma classe abstrata chamada “Animal”, tendo uma relação de herança com as outras classes presentes

no código. Sempre que queremos criar uma classe abstrata, devemos utilizar o comando `abstract class`, assim o java compreenderá que aquele elemento é uma classe abstrata, que pode ser herdada por outras classes concretas, mas jamais instanciada.

61 - Métodos Abstratos

Métodos abstratos são métodos que não possuem corpo, apenas assinatura e esses métodos devem ser obrigatoriamente implementados dentro de suas classes filhas com o objetivo de definir o comportamento específico de cada uma. Como por exemplo:

```
public abstract class Animal {
    abstract void formaDeMover();
}

public class Baleia extends Animal{
    @Override
    public void formaDeMover(){
        System.out.println("Nadando");
    }
}

public class Cachorro extends Animal{
    @Override
    public void formaDeMover(){
        System.out.println("Andando");
    }
}

public class papagaio extends Animal{
    @Override
    public void formaDeMover() {
        System.out.println("Voando");
    }
}
```

No exemplo acima, podemos observar a criação de um método abstrato `abstract void formaDeMover();` dentro de uma classe abstrata. Como

afirmamos anteriormente, este método não possui conteúdo, mas deve obrigatoriamente ser implementado em suas subclasses com suas especificações.

Neste caso em específico, todos os animais possuem suas formas de se mover: a baleia nada; o cachorro anda; o papagaio voa; implementando o conceito abstrato de “Se mover”.

6.2 - Classes abstratas x Classes Concretas

Classes Abstratas	Classes Concretas
Não pode ser instanciada, mas pode estabelecer relação de **herança**.	Pode ser **instanciada** e pode estabelecer relação de **herança**.
Pode receber métodos abstratos e métodos concretos (Com implementação).	Pode conter **apenas** métodos concretos, com implementação.