Interpreter Padroes De Projeto Em Java

Inúmeras vezes vamos precisar escrever códigos que atendam a nossa necessidade, porém, muitas vezes outras pessoas já passaram pelos mesmos problemas e criaram uma solução, algumas soluções são tão boas que outros seguiram a mesma, tornando assim, uma solução padrão para um determinado problema ou necessidade.

Assim criou-se os padrões de projeto, também conhecidos como *design patterns*, para o exemplo do *post* irei mostrar o padrão chamado Interpreter (Interpretador) com ele vamos criar uma forma para interpretar expressões matemáticas para as quatro operações básicas, sendo elas: Adição, subtração, multiplicação e divisão.

Entendendo uma expressão matemática

Antes de partirmos para os códigos, precisamos entender como funciona uma expressão matemática. Para formar uma expressão sempre iremos precisar de no mínimo dois números, um á esquerda e outro á direita, entre os números deve vim um operador matemático (`+` , `-` , `*` ou `/`). Sabendo disso, já podemos criar algumas expressões:

```
1+1
3-2
3*4
10/5
```

Mas as expressões não limitam-se apenas a números, também podemos ter uma expressão que utiliza o resultado de outra expressão, por exemplo:

```
(1+1)*10
(9-5)+(10-2)
(10/2)-2
```

Legal, já entendemos como montar nossas expressões, porém, a questão é:

> Como fazer para criar essa calculadora de expressões utilizando programação orientada á objetos?

Exatamente por isso, vamos utilizar o padrão de projeto Interpreter.

Conhecendo o padrão Interpreter

O Interpreter é um padrão responsável por criar interpretações de código, ou seja, passamos algo para ele e o mesmo será responsável por processar e interpretar nosso parâmetro, ele é muito utilizando para interpretar DSL's ou criar compiladores.

Com ele, vamos criar nossas expressões e pedir para que realize a interpretação, ou seja, ele irá realizar o cálculo da expressão matemática passada.

Criando nosso interpretador

Legal, já temos o necessário para começar a criar nosso interpretador de expressões matemáticas, mas, por onde devemos começar? Se vamos realizar operações matemáticas, vamos precisar de um número, então, devemos começar a sua criação:

```
package br.com.matheuscastiglioni.interpreter;

public class Numero {
}
```

Repare que criamos a classe `Numero` dentro do pacote `br.com.matheuscastiglioni.interpreter`.

Legal, agora, o que nosso objeto número deve receber? Precisamos passar para ele, o número de fato que queremos utilizar na expressão.

```
package br.com.matheuscastiglioni.interpreter;

public class Numero {

    private int numero;

    public Numero(int numero) {
        this.numero = numero;
    }

}
```

Pronto, agora nosso construtor está esperando um parâmetro númerico.

Por questões de simplicidade vamos utilizar apenas números inteiros.

Legal, já temos nosso número, agora podemos criar nosso primeiro operador matemático, vamos começar pela adição:

```
package br.com.matheuscastiglioni.interpreter;

public class Somar {
}
```

Criamos a classe `Somar`, ela será responsável por interpressar expressões de adição. Sabendo que uma expressão deve receber um número á direita e outro á esquerda, podemos ver que são dois fortes candidatos á serem recebidos em nosso constructor, pois, sem eles não iremos ter o que interpretar.

```
package br.com.matheuscastiglioni.interpreter;

public class Somar {

    private Numero esquerda;
    private Numero direita;

    public Somar(Numero esquerda, Numero direita) {
        this.esquerda = esquerda;
        this.direita = direita;
    }

}
```

Pronto, tudo certo, agora já temos o suficiente para interpretar nossa expressão de adição, precisamos apenas criar um método que faça isso:

```
package br.com.matheuscastiglioni.interpreter;

public class Somar {

    private Numero esquerda;
    private Numero direita;
```

```
    public Somar(Numero esquerda, Numero direita) {
        this.esquerda = esquerda;
        this.direita = direita;
    }

    public int interpretar() {
        return this.esquerda + this.direita;
    }

}
```

Se fizermos assim, o código não irá compilar. Porque isso está acontecendo? Repare que nossos parâmetros são do tipo `Numero`, em outras palavras, é um objeto, será que um objeto em Java sabe e/ou pode ser somado com outro objeto? Não, isso é impossível. Então, precisamos de alguma forma dizer para o `Numero` que ele deve se interpretar, devolvendo o seu valor, para depois utilizar esse resultado e interpretar a soma.

Criando nossa interface

Sabemos que no Java podemos utilizar uma `Interface` para dizer como uma classe deve se comportar, ou seja, quais métodos será obrigatório que ela tenha. Chamamos a `interface` de contrato e quem assiná-lo deve seguir suas regras.

```
package br.com.matheuscastiglioni.interpreter;

public interface Operador {
}
```

Criamos a interface `Operador`, pois até o momento tudo é um operador, seja numérico ou matemático. Agora precisamos dizer que todo `Operador` deve saber se `interpretar`.

```
package br.com.matheuscastiglioni.interpreter;

public interface Operador {

    int interpretar();

}
```

Assinando nosso contrato

Legal, criamos o contrato e quem assiná-lo se tornará um `Operador` e deve saber se interpretar. Precisamos agora primeiramente fazer com que nosso número assine esse contrato, podemos fazer isso utilizando a palavra `implements` e passando qual `interface` ele deve implementar (assinar).

```
package br.com.matheuscastiglioni.interpreter;

public class Numero implements Operador {

    private int numero;

    public Numero(int numero) {
        this.numero = numero;
    }

}
```

Apenas essa modificação não será suficiente, nosso código não deve compilar, lembra quando foi falado que: "Um contrato assinado deve ser seguido", sendo assim, precisamos escrever nosso método `interpretar` pois é a única regra que existe em nosso contrato.

```
package br.com.matheuscastiglioni.interpreter;

public class Numero implements Operador {

    private int numero;

    public Numero(int numero) {
        this.numero = numero;
    }

    @Override
    public int interpretar() {
        return this.numero;
    }
}
```

Veja que a interpretação de um número é ele mesmo.

Interpretando nossa adição

Legal, até o momento nossa classe `Somar` está da seguinte maneira:

```
package br.com.matheuscastiglioni.interpreter;

public class Somar {

    private Numero esquerda;
    private Numero direita;

    public Somar(Numero esquerda, Numero direita) {
        this.esquerda = esquerda;
        this.direita = direita;
    }

    public int interpretar() {
        return this.esquerda + this.direita;
    }

}
```

Ela ainda não compila, o primeiro passo será assinar o contrato:

```
package br.com.matheuscastiglioni.interpreter;

public class Somar implements Operador {

    private Numero esquerda;
    private Numero direita;

    public Somar(Numero esquerda, Numero direita) {
```

```
        this.esquerda = esquerda;
        this.direita = direita;
    }

    public int interpretar() {
        return this.esquerda + this.direita;
    }

}
```

Como a classe `Somar` já tinha um método chamado `interpretar` não foi necessário escrevê-lo, mas, para deixar explicíto que o método está sendo criado devido ao nosso contrato, vamos adicionar a anotação `@Override` nele.

```
package br.com.matheuscastiglioni.interpreter;

public class Somar implements Operador {

    private Numero esquerda;
    private Numero direita;

    public Somar(Numero esquerda, Numero direita) {
        this.esquerda = esquerda;
        this.direita = direita;
    }

    @Override
    public int interpretar() {
        return this.esquerda + this.direita;
    }

}
```

Assim quem ler o código saberá que o método `interpretar` está sendo sobreescrito da `interface` `Operador`, ou seja, estamos seguindo a regra do contrato.

Legal, agora já sabemos que podemos pegar o valor de nosso `Numero` através do método `interpretar`, então vamos utilizá-lo:

```
package br.com.matheuscastiglioni.interpreter;

public class Somar implements Operador {

    private Numero esquerda;
    private Numero direita;

    public Somar(Numero esquerda, Numero direita) {
        this.esquerda = esquerda;
        this.direita = direita;
    }

    @Override
    public int interpretar() {
        return this.esquerda.interpretar() + this.direita.interpretar();
    }

}
```

Repare que estamos recebendo dois números em nosso construtor, mas, no começo do *post* vimos que uma expressão pode ser composta por outra, sendo assim, poderíamos passar o resultado de uma divisão como parâmetro, mas, dessa maneira não seremos capazes, porque estamos esperando apenas números como parâmetros, como podemos resolver isso?

Pense bem, criamos o nosso contrato para ter certeza que, **"Quem assiná-lo deve seguir suas regras"**, sendo assim, toda classe que implementá-lo deve sobrescrever o método `interpretar`, isso, nos dá uma certeza.

Será que não podemos receber nosso contrato como parâmetro da classe `Somar`? Sim, podemos e é exatamente dessa maneira que vamos resolver o problema:

```
package br.com.matheuscastiglioni.interpreter;

public class Somar implements Operador {

    private Operador esquerda;
    private Operador direita;

    public Somar(Operador esquerda, Operador direita) {
        this.esquerda = esquerda;
        this.direita = direita;
    }

    @Override
    public int interpretar() {
        return this.esquerda.interpretar() + this.direita.interpretar();
    }

}
```

Trocamos o tipo de nossos atribútos e parâmetros para `Operador`, assim, podemos passar um operador de `Dividir`, `Multiplicar` ou `Subtrair` (classes que iremos criar da mesma maneira que a `Somar`, mudando apenas o operador matemático):

Subtrair

```
package br.com.matheuscastiglioni.interpreter;

public class Subtrair implements Operador {

    private Operador esquerda;
    private Operador direita;

    public Subtrair(Operador esquerda, Operador direita) {
        this.esquerda = esquerda;
        this.direita = direita;
    }

    @Override
    public int interpretar() {
        return this.esquerda.interpretar() - this.direita.interpretar();
    }

}
```

Multiplicar

```
package br.com.matheuscastiglioni.interpreter;

public class Multiplicar implements Operador {

    private Operador esquerda;
    private Operador direita;

    public Multiplicar(Operador esquerda, Operador direita) {
        this.esquerda = esquerda;
        this.direita = direita;
    }

    @Override
    public int interpretar() {
        return this.esquerda.interpretar() * this.direita.interpretar();
    }

}
```

Dividir

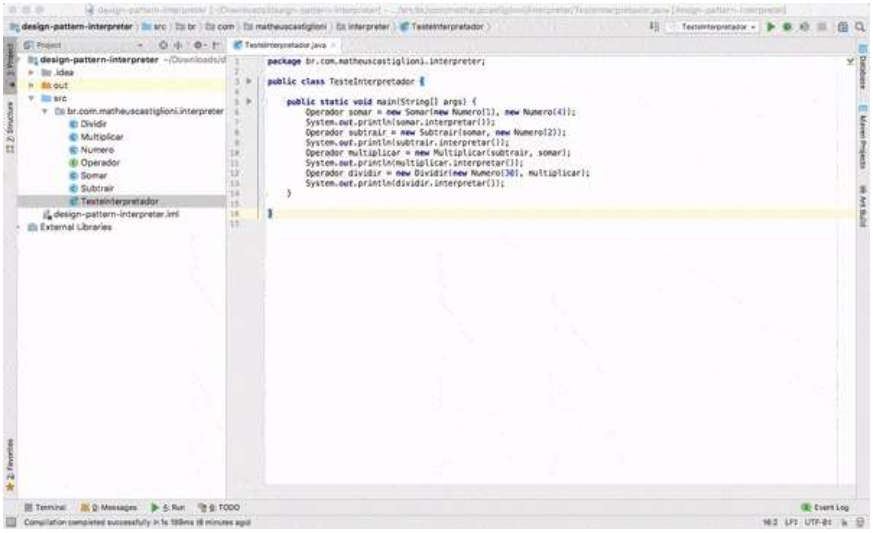
```
package br.com.matheuscastiglioni.interpreter;

public class Dividir implements Operador {

    private Operador esquerda;
    private Operador direita;

    public Dividir(Operador esquerda, Operador direita) {
        this.esquerda = esquerda;
        this.direita = direita;
    }

    @Override
    public int interpretar() {
        return this.esquerda.interpretar() / this.direita.interpretar();
    }

}
```

Pronto, já temos nossos números e operadores prontos, podemos testá-los:

```
package br.com.matheuscastiglioni.interpreter;

public class TesteInterpretador {

    public static void main(String[] args) {
        Operador somar = new Somar(new Numero(1), new Numero(4));
        System.out.println(somar.interpretar());
        Operador subtrair = new Subtrair(somar, new Numero(2));
        System.out.println(subtrair.interpretar());
        Operador multiplicar = new Multiplicar(subtrair, somar);
        System.out.println(multiplicar.interpretar());
```

```
        Operador dividir = new Dividir(new Numero(30), multiplicar);
        System.out.println(dividir.interpretar());
    }

}
```

Rodando nossa classe de teste, temos o seguinte resultado:



Repare que as saídas foram:

- **5**: 1 + 4 = 5
- **3**: Resultado da soma anterior (5) - 2 = 3
- **15**: Resultado da subtração anterior (3) * resultado da soma (5) = 15
- **2**: 30 / resultado da multiplicação anterior (15) = 2

Show de bola, tudo funcionando corretamente.

Saiba mais

Além do padrão Interpreter, já mostrei no blog o padrão Strategy onde criei uma Calculadora de Impostos no post Strategy - Padrões de Projeto em Java não deixe de conferir, nele, explico como realizar cálculos de diversos impostos sem a utilização de `ifs` .

Conclusão

Nesse *post* expliquei como criar um interpretador de expressões matemáticas utilizando o padrão de projeto **Interpreter**.

Espero que tenha gostado, não deixe de comentar e assinar a *newsletter* para receber novidades por email.

O projeto do *post* pode ser encontrado aqui.

Até a próxima \o/

Matheus Castiglioni

Apaixonado pelo mundo dos códigos e um eterno estudante, gosto de aprender e saber um pouco de tudo, aquela curiosidade de saber como tudo funciona, tento compartilhar o máximo de conhecimentos adquiridos e ajudar todos aqueles que sou capaz.

4 COMENTÁRIOS **Blog do Matheus Castiglioni** **Disqus' Privacy Policy** **Iniciar sessão**

Recomendar Tweet Partilhar **Mostrar primeiro os mais recentes**

Escreva o seu comentário...

INICIE SESSÃO COM O **OU REGISTE-SE NO DISQUS** ?

Nome

Gabriel • há um ano
Melhor explicação
Responder • Partilhar ›

Matheus Castiglioni Moderador Gabriel • há um ano
Obrigado Gabriel, ainda bem que ficou boa \o/
Responder • Partilhar ›

Marlysson Silva • há 2 anos
Eita.. show hein, bem explicado.
Responder • Partilhar ›

Matheus Castiglioni Moderador Marlysson Silva • há 2 anos
Obrigado Marlysson, fico feliz que tenha gostado.
Responder • Partilhar ›

Subscrever **Acerca do DisqusAdicionar o DisqusAdicionar** **Do Not Sell My Data**

Feito com ♥ por **Matheus Castiglioni** com **Hugo**.