Herança, Polimorfismo e Interfaces

Herança

A **Herança** permite que uma classe (filha, ou subclasse) reutilize os atributos e métodos de outra classe (pai, ou superclasse).

- Reutiliza código evitando duplicação
- Facilita manutenção
- Cria uma hierarquia de classes

Se pensarmos no exemplo dos funcionários de uma empresa, podemos citar entre eles algumas características em comum, porém algumas diferenças específicas entre Gerentes e Desenvolvedores, o que nos faria representá-los com diferentes subclasses.

Herança

Para representar uma herança em Java, usamos a palavra reservada **extends**.

```
public class Gerente extends Funcionario
```

Com isso, garantimos que a subclasse **Gerente** terá todos os atributos e métodos definidos na superclasse **Funcionario**.

Usa-se herança quando há uma relação clara de **"é um"** (Gerente **É UM** Funcionario). É importante evitar heranças profundas (muitas camadas), para não deixar o código muito complexo.

Polimorfismo

O **polimorfismo** permite que um mesmo método tenha comportamentos diferentes dependendo da classe que o implementa. Por exemplo, em nosso *case* de funcionários de uma empresa, poderíamos ter um método **calcularBonificacao** que fosse diferente para os diversos tipos de funcionário.

Poderia corresponder a 1% do salário para funcionários em geral, mas para o gerente poderia ser um valor definido no atributo **bonus** que ele deve receber todo mês. Para isso teríamos, na classe **Funcionario**:

```
public double calcularBonificacao() {
    return salario * 0.01;
}
```

Polimorfismo

Já na classe Gerente, poderíamos implementar o método da seguinte forma:

```
@Override
public double calcularBonificacao() {
    return bonus;
}
```

Com isso, ao chamar o método **calcularBonificacao** para uma instância de **Funcionario**, caso a mesma fosse de um **Gerente**, o cálculo realizado seria o que está descrito acima.

Polimorfismo

Com isso, temos:

- Flexibilidade → Chamadas genéricas para objetos diferentes
- Extensibilidade → Facilidade para adicionar novas funcionalidades e comportamentos

Existem dois tipos de polimorfismo: o polimorfismo de sobrescrita (*Override*) e de sobrecarga (*Overload*).

O de sobrescrita é quando uma subclasse redefine um método da subclasse. É exatamente o exemplo que acabamos de ver, onde o método **calcularBonificacao** tem um comportamento diferente para a subclasse Gerente.

Polimorfismo

O polimorfismo de sobrecarga é quando temos métodos com o mesmo nome, mas **parâmetros** diferentes (ou seja, assinaturas diferentes).

Exemplo: poderíamos ter um outro método chamado **calcularBonificacao**, onde fosse passado o parâmetro do percentual, para utilizar um percentual específico no cálculo. Dessa forma, de acordo com a chamada (com ou sem percentual), um ou outro método seria executado.

```
public double calcularBonificacao(double percentual) {
    return salario * (percentual / 100);
}
```

Polimorfismo

O polimorfismo de sobrecarga é quando temos métodos com o mesmo nome, mas **parâmetros** diferentes (ou seja, assinaturas diferentes).

Exemplo: poderíamos ter um outro método chamado **calcularBonificacao**, onde fosse passado o parâmetro do percentual, para utilizar um percentual específico no cálculo. Dessa forma, de acordo com a chamada (com ou sem percentual), um ou outro método seria executado.

```
public double calcularBonificacao(double percentual) {
    return salario * (percentual / 100);
}
```

Classes e métodos abstratos

Em alguns casos, podemos querer que uma superclasse tenha um comportamento apenas de modelo, para centralizar atributos e métodos em comum, sem desejar que ela seja instanciada diretamente. Para isso, podemos definir a classe como **abstrata**. Utilizamos a palavra reservada **abstract** em Java.

```
public abstract class Funcionario
```

Existe a possibilidade de definir apenas os métodos também como abstratos, para obrigar que os mesmos sejam implementados pelas subclasses. Ao fazer isso, TODAS as subclasses deverão implementar o método. Exemplo de método definido na classe Funcionario.

```
public abstract void calcularPLR();
```

Interfaces

Uma interface define um **contrato** que as classes devem seguir, mas não implementa métodos, apena os define. Isso promove:

- Desacoplamento → Separa definição da implementação
- Flexibilidade → Facilita substituição de classes
- Autonomia → Permite múltipla implementação

Por exemplo: uma empresa pode ter alguns projetos e apenas alguns tipos de funcionários podem aprovar a execução dos mesmo, seguindo regras diferentes. Poderíamos criar uma interface denominada **Aprovador**, com o método *aprovarProjeto*, implementando-a diferentemente para cada tipo de funcionário.

Interfaces

A definição da interface ficaria, por exemplo, como abaixo:

```
public interface Aprovador {
    void aprovarProjeto(String nomeProjeto);
}
```

E na classe Gerente, para definir que esse método deverá ser implementado, utilizaremos a palavra reservada **implements**.

```
public class Gerente extends Funcionario implements Aprovador
```

Interfaces

Ao usar a palavra reservada *implements*, somos automaticamente obrigados a implementar **todos** os métodos definidos na interface.

```
@Override
public void aprovarProjeto(String nomeProjeto) {
    //implementação aqui
}
```

Java não permite herança múltipla, mas permite a implementação de várias interfaces, o que favorece a definição de comportamentos comuns entre classes diferentes.

Compartilhe um resumo de seus novos conhecimentos em suas redes sociais.
#aprendizadoalura

alura | **Escola** Programação