ENTENDENDO POLIMORFISMO

Polimorfismo em Java: o que é, pra que serve, como e onde usar

Agora que aprendemos os conceitos mais importantes e vimos o **uso da Herança em Java**, vamos estudar outra características marcante da programação Java e, de uma maneira mais geral, da programação orientada a objetos: o **polimorfismo**.

Com o polimorfismo vamos ter um controle maior sobre as subclasses sem ter que nos preocupar especificamente com cada uma delas, pois cada uma terá autonomia para agir de uma maneira diferente.

Definição de polimorfismo em Java

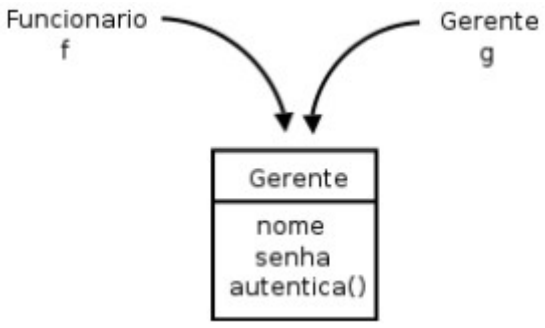
Traduzindo, do grego, ao pé da letra, polimorfismo significa "muitas formas".
Essas formas, em nosso contexto de programação, são as subclasses/objetos criados a partir de uma classe maior, mais geral, ou abstrata.
Polimorfismo é a capacidade que o Java nos dá de controlar todas as formas de uma maneira mais simples e geral, sem ter que se preocupar com cada objeto especificamente.

Mais uma vez, somente pela definição é muito complicado de entender.
Vamos partir para algo comum e de preocupação máxima de nosso **curso de Java** : exemplos práticos.

O que guarda uma variável do tipo Funcionario ? Uma referência para um Funcionario , nunca o objeto em si.

Na herança, vimos que todo Gerente é um Funcionario , pois é uma extensão deste. Podemos nos referir a um Gerente como sendo um Funcionario . Se alguém precisa falar com um Funcionario do banco, pode falar com um Gerente ! Porque? Pois Gerente é um Funcionario . Essa é a semântica da herança.

```
Gerente gerente = new Gerente();
Funcionario funcionario = gerente;
funcionario.setSalario(5000.0);
```



Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).

Até aqui tudo bem, mas e se eu tentar
`funcionario.getBonificacao();`

Qual é o retorno desse método? 500 ou 750? No Java, a invocação de método sempre vai ser decidida em tempo de execução. O Java vai procurar o objeto na memória e, aí sim, decidir qual método deve ser chamado, sempre relacionando com sua classe de verdade, e não com a que estamos usando para referenciálo. Apesar de estarmos nos referenciando a esse Gerente como sendo um Funcionario, o método executado é o do Gerente. O retorno é 750. Parece estranho criar um gerente e referenciá-lo como apenas um funcionário. Por que faríamos isso? Na verdade, a situação que costuma aparecer é a que temos um método que recebe um argumento do tipo Funcionario:

Parece estranho criar um gerente e referenciá-lo como apenas um funcionário. Por que faríamos isso? Na verdade, a situação que costuma aparecer é a que temos um método que recebe um argumento do tipo *Funcionario*:

```
class ControleDeBonificacoes {
	private double totalDeBonificacoes = 0;

	public void registra(Funcionario funcionario) {
		this.totalDeBonificacoes += funcionario.getBonificacao();
	}

	public double getTotalDeBonificacoes() {
		return this.totalDeBonificacoes;
	}
}
```

E, em algum lugar da minha aplicação (ou no main, se for apenas para testes):

```
ControleDeBonificacoes controle = new ControleDeBonificacoes();

Gerente funcionario1 = new Gerente(); funcionario1.setSalario(5000.0);
controle.registra(funcionario1);

Funcionario funcionario2 = new Funcionario(); funcionario2.setSalario(1000.0);
controle.registra(funcionario2);

System.out.println(controle.getTotalDeBonificacoes());
```

Repare que conseguimos passar um Gerente para um método que recebe um Funcionario como argumento. Pense como numa porta na agência bancária com o seguinte aviso: “Permitida a entrada apenas de Funcionários”. Um gerente pode passar nessa porta? Sim, pois Gerente é **um** Funcionario.

Qual será o valor resultante? Não importa que dentro do método registra do ControleDeBonificacoes receba Funcionario. Quando ele receber um objeto que realmente é um Gerente, o seu método reescrito será invocado. Reafirmando: **não importa como nos referenciamos a um objeto, o método que será invocado é sempre o que é dele**.

No dia em que criarmos uma classe Secretaria, por exemplo, que é lha de Funcionario, precisaremos mudar a classe de ControleDeBonificacoes? Não. Basta a classe Secretaria reescrever os métodos que lhe parecerem necessários. É exatamente esse o poder do polimor smo, juntamente com a reescrita de método: diminuir o acoplamento entre as classes, para evitar que novos códigos resultem em modificações em inúmeros lugares.

Repare que quem criou ControleDeBonificacoes pode nunca ter imaginado a criação da classe Secretaria ou Engenheiro. Contudo, não será necessário reimplementar esse controle em cada nova classe: reaproveitamos aquele código.

Herança versus acoplamento

Note que o uso de herança aumenta o acoplamento entre as classes, isto é, o quanto uma classe depende de outra. A relação entre classe mãe e filha é muito forte e isso acaba fazendo com que o programador das classes filhas tenha que conhecer a implementação da classe pai e vice-versa - fica difícil fazer uma mudança pontual no sistema.

Por exemplo, imagine se tivermos que mudar algo na nossa classe Funcionario, mas não quiséssemos que todos os funcionários sofressem a mesma mudança. Precisaríamos passar por cada uma das filhas de Funcionario verificando se ela se comporta como deveria ou se devemos sobreescrever o tal método modificado.

Esse é um problema da herança, e não do polimorfismo, que resolveremos mais tarde com a ajuda de Interfaces.

Exemplo:

Imagine que vamos modelar um sistema para a faculdade que controle as despesas com funcionários e professores. Nosso funcionário fica assim:

```
class EmpregadoDaFaculdade {
	private String nome;
	private double salario;
	double getGastos() {
		return this.salario;
	}
	String getInfo() {
		return "nome: " + this.nome + " com salário " + this.salario;
	}
	// métodos de get, set e outros
}
```

O gasto que temos com o professor não é apenas seu salário. Temos de somar um bônus de 10 reais por hora/aula. O que fazemos então? Reescrevemos o método. Assim como o `getGastos` é diferente, o `getInfo` também será, pois temos de mostrar as horas/aula também.

```
class ProfessorDaFaculdade extends EmpregadoDaFaculdade {
private int horasDeAula; double getGastos() { return this.getSalario() + this.horasDeAula * 10; }
String getInfo() { String informacaoBasica = super.getInfo(); String informacao =
informacaoBasica + " horas de aula: " + this.horasDeAula; return informacao; } // métodos de get,
set e outros que forem necessários }
```

A novidade, aqui, é a palavra chave super. Apesar do método ter sido reescrito, gostaríamos de acessar o método da classe mãe, para não ter de copiar e colocar o conteúdo desse método e depois concatenar com a informação das horas de aula.