POLIMORFISMO

O encapsulamento é o primeiro pilar da programação orientada a objetos, a herança é o segundo pilar e **o polimorfismo é o terceiro pilar.** O polimorfismo permite criar software que facilite a sua futura alteração e manutenção.

Polimorfismo significa muitas formas. Em programação orientada a objetos o polimorfismo permite que um único nome de classe ou nome de método represente um código diferente, selecionado por algum mecanismo automático. Assim, como o mesmo nome pode representar código diferente, o mesmo nome pode representar muitos comportamentos diferentes.

Para entendermos melhor o polimorfismo vamos ver o exemplo abaixo. Temos uma classe Funcionario que possui um método getBonificacao() que calcula e retorna uma bonificação de 10% do salário do Funcionário:

```
public class Funcionario {

   protected String nome;
   protected String cpf;
   protected double salario;

   public double getBonificacao() {
      return this.salario * 0.10;
   }

   public String getNome() {
      return nome;
   }

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

   public String getCpf() {
      return cpf;
   }

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

   public double getSalario() {
      return salario;
   }

   public void setSalario(double salario) {
      this.salario = salario;
   }
}
```

Depois temos uma classe Gerente que é uma subclasse de funcionário em que o código é apresentado a seguir.

```
public class Gerente extends Funcionario {
  private int senha;
  private int numeroDeFuncionariosGerenciados;

   public double getBonificacao() {
       return this.salario * 0.15;
   }

  public boolean autentica(int senha) {
    if (this.getSenha() == senha) {
        System.out.println("Acesso Permitido!");
        return true;
    } else {
        System.out.println("Acesso Negado!");
        return false;
    }
  }

    public int getSenha() {
       return senha;
    }

    public void setSenha(int senha) {
       this.senha = senha;
    }

    public int getNumeroDeFuncionariosGerenciados() {
       return numeroDeFuncionariosGerenciados;
    }

    public void setNumeroDeFuncionariosGerenciados(int numeroDeFuncionariosGerenciados) {
       this.numeroDeFuncionariosGerenciados = numeroDeFuncionariosGerenciados;
    }
}
```

A classe Gerente tem um atributo senha e outro atributo numeroDeFuncionariosGerenciados e tem um método que autentica() que vai retornar verdadeiro se a senha passada está correta ou falso se a senha passada estiver incorreta.

O que é importante perceber neste exemplo é que o método getBonificacao() que existe na superclasse Funcionario é sobrescrito aqui na classe Gerente. Isto pois o cálculo da bonificação do Gerente é diferente do cálculo da bonificação do Funcionário. O gerente recebe uma bonificação de 15% e o funcionário de 10%.

Pela herança, sabe-se que neste exemplo 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. Esse é o conceito da herança.

Assim, podemos criar uma aplicação no método main() que instancia um objeto da classe Funcionário e outro da classe Gerente para testar o funcionamento destas classes de acordo com o código a seguir.

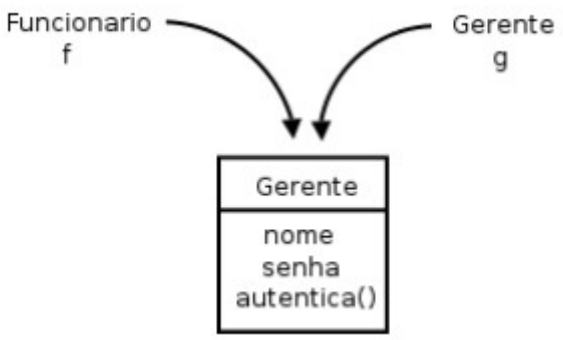
```
public static void main(String[] args) {

        Funcionario f1 = new Funcionario();
        f1.setNome("Jose Antonio");
        f1.setCpf("111111111");
        f1.setSalario(1000);
        System.out.println("Funcionario");
        System.out.println("Nome: "+f1.getNome());
        System.out.println("Cpf: "+ f1.getCpf());
        System.out.println("Salário: " + f1.getSalario());
        System.out.println("Bonificação: " + f1.getBonificacao());

        Gerente f2 = new Gerente();
        f2.setNome("Pedro Henrique");
        f2.setCpf("222222222");
        f2.setSenha(4321);
        f2.setSalario(5000);
        if (f2.autentica(4321)) {
            System.out.println("Gerente");
            System.out.println("Nome: "+f2.getNome());
            System.out.println("Cpf: "+ f2.getCpf());
            System.out.println("Salário: " + f2.getSalario());
            System.out.println("Bonificação: " + f2.getBonificacao());
        } else {
           System.out.println("A senha está incorreta");
        }
}
```

Se fossemos criar o seguinte código no final do método main() anterior estaríamos criando um polimorfismo:

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



Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. E no exemplo acima temos o mesmo objeto tendo duas referências (nomes) que são gerente e funcionario. A primeira linha mostra que foi criado (instanciado) um objeto da classe Gerente que tem a referência chamada gerente. Na segunda linha foi criada uma referência chamada funcionario

da classe Funcionario mas que aponta (referencia) para o mesmo objeto gerente. Isto é possível pois pela herança um Gerente é um Funcionario.

Mas e se nós após o código anterior chamássemos o seguinte método?

funcionario.getBonificacao();

O Java vai execuar o método getBonificacao() da classe Funcionario ou da classe Gerente ? 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 referenciá-lo. Apesar de estarmos nos referenciando a esse Gerente como sendo um Funcionario, o método executado é o do Gerente.

Mas para entendermos ainda melhor o polimorfismo vamos imaginar que precisamos criar uma outra classe que totalize e guarde a soma das bonificações de vários funcionários e gerentes. Para isso criaríamos a seguinte classe:

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

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

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

Criamos o método registra() da classe ControleDeBonificacoes que faz o polimorfismo (é polimórfico) pois ele recebe como parâmetro um objeto da classe Funcionario e vai calcular e retornar a bonificação do objeto que ele receber. Assim, o método é polimórfico pois tem o mesmo nome registra() e vai ter comportamentos diferentes pois caso ele receber um objeto da classe Funcionário ele calcula a bonificação de 10% e se ele receber um objeto da classe Gerente (que também é um funcionário) o mesmo método vai calcular a bonificação de 15%.

No dia em que criarmos uma classe Secretária, por exemplo, que é filha de Funcionario, precisaremos mudar a classe de ControleDeBonificacoes? Não. Basta a classe Secretária reescrever os métodos que lhe parecerem necessários. É exatamente esse o poder do polimorfismo, 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 Secretária ou Engenheiro. Contudo, não será necessário reimplementar esse controle em cada nova classe: reaproveitamos aquele código.

Assim, poderíamos criar o seguinte código no final do método main() anterior para testarmos a classe ControleDeBonificacoes:

```
ControleDeBonificacoes controle = new ControleDeBonificacoes();
controle.registra(f1);
controle.registra(f2);
System.out.println( "O total de bonificações é:" +controle.getTotalDeBonificacoes());
```