

Fabio de Moura Camargo Neto
*github.com/bio353*

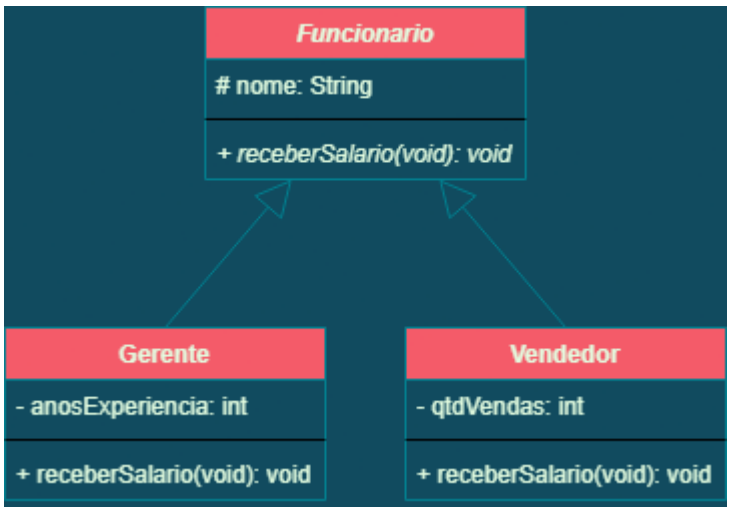
# Polimorfismo e Reescrita

## Introdução

### Definição

- Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse.
- Do grego, significa “mais de uma forma” (*poli* = muitas, *morphos* = formas*)*, ou seja, o mesmo objeto realizando diferentes operações de acordo com a forma que foram requisitadas.

### Exemplo




1. Na imagem acima, temos três classes: `Funcionario` (abstrata), `Gerente` e `Vendedor`.
2. Todos os funcionários precisam ser pagos ao final do mês, ou seja, todos devem possuir um método `receberSalario()`.
3. Entretanto, cada funcionário recebe o salário de uma maneira diferente:
    a. `Gerente` recebe o salário com base em um valor fixo de 3 salários mínimos, somado a um valor calculado com base nos anos de experiência (`anosExperiencia`) trabalhando na empresa;
    b. `Vendedor` recebe o salário com base em um valor fixo de 1 salário mínimo, somado a um valor calculado com base na quantidade de vendas ou comissões (`qtdVendas`) feitas naquele mês.
4. Perceba que os métodos, implementados em `Gerente` e em `Vendedor` possuem mesmo nome, ou seja, a mesma chamada, mas comportamentos distintos.
5. Este é um típico caso de **polimorfismo**!

# Sobreposição e Sobrecarga de Métodos

## Polimorfismo Dinâmico ou Sobreposição de Métodos

- Uma das formas de implementar o polimorfismo é através de uma classe abstrata, cujos métodos são declarados, mas não são definidos, e através de classes que herdam os métodos desta mesma classe.
    - É o tipo mais comum quando falamos sobre polimorfismo.

    - O exemplo anterior é um caso de sobreposição de métodos.
- Em **Java** ☕, ela é feita através da utilização da anotação `@Override`. Veja o exemplo:

```
public abstract class Funcionario {
    protected String nome;

    public abstract void receberSalario();
}

public class Gerente extends Funcionario {
    private int anosExperiencia;

    Gerente(int anosExperiencia, String nome) {
        this.nome = nome;
        this.anosExperiencia = anosExperiencia;
    }

    @Override
    public void receberSalario() {
        float salario = (float) (3636.03 + anosExperiencia * 100);
        System.out.println(nome + " recebe R$" + salario + " mensais.");
    }
}

public class Vendedor extends Funcionario {
    private int qtdVendas;

    Vendedor(int qtdVendas, String nome) {
        this.nome = nome;
        this.qtdVendas = qtdVendas;
    }

    @Override
    public void receberSalario() {
        float salario = (float) (1212.01 + qtdVendas * 20);
        System.out.println(nome + " recebe R$" + salario + " mensais.");
    }
}

public class Empresa {
    public static void main(String[] args) {
        Gerente g1 = new Gerente(10, "Menino Ney");
        Vendedor v1 = new Vendedor(7, "Rainha Elizabeth");
        g1.receberSalario();
        v1.receberSalario();
    }
}
```

```
Menino Ney recebe R$4636.03 mensais.
Rainha Elizabeth recebe R$1352.01 mensais.
```

## Polimorfismo Estático ou Sobrecarga de Métodos

- Métodos de mesmo nome, mas com parâmetros distintos.
- Diferenciado em tempo de execução e não de compilação.
- Criemos um segundo construtor, onde `nome` não é passado a ele. Por boa prática, inicializaremos `nome` de maneira maneira genérica.

```
public class Vendedor extends Funcionario {
    private int qtdVendas;

    Vendedor(int qtdVendas, String nome) {
        this.nome = nome;
        this.qtdVendas = qtdVendas;
    }

    Vendedor(int qtdVendas) {
        this.nome = "Vendedor(a)";
        this.qtdVendas = qtdVendas;
    }

    @Override
    public void receberSalario() {
        float salario = (float) (1212.01 + qtdVendas * 20);
        System.out.println(nome + " recebe R$" + salario + " mensais.");
    }
}

public class Empresa {
    public static void main(String[] args) {
        Gerente g1 = new Gerente(10, "Menino Ney");
        Vendedor v1 = new Vendedor(7, "Rainha Elizabeth");
        Vendedor v2 = new Vendedor(5);
        g1.receberSalario();
        v1.receberSalario();
        v2.receberSalario();
    }
}
```

```
Menino Ney recebe R$4636.03 mensais.
Rainha Elizabeth recebe R$1352.01 mensais.
Vendedor(a) recebe R$1312.01 mensais.
```

# Referenciando Objetos

- Em **Java** ☕, também é possível fazer referência a um objeto de uma classe filha através de uma classe pai. Por exemplo:

```
public class Empresa {
    public static void main(String[] args) {
        Gerente g1 = new Gerente(10, "Menino Ney");
        Vendedor v1 = new Vendedor(7, "Rainha Elizabeth");
        Vendedor v2 = new Vendedor(5);
        Funcionario[] funcionarios = {g1, v1, v2};
        funcionarios[0].receberSalario();
        funcionarios[0].receberSalario();
        funcionarios[0].receberSalario();
    }
}
```

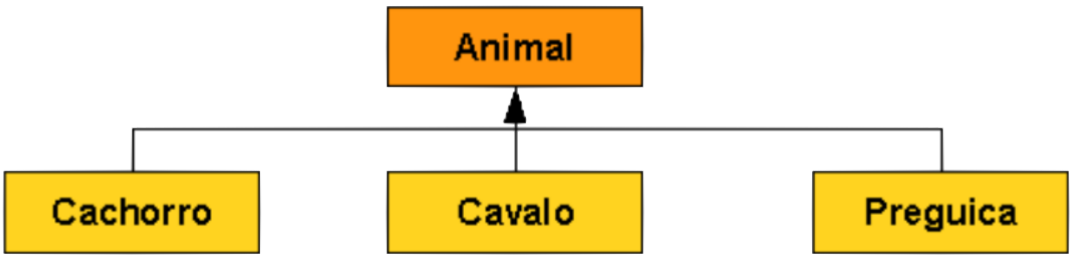
- Desta forma, podemos criar uma lista de objetos do tipo `Funcionario` e depois chamarmos o método `receberSalario()` dentro de um `for` *loop*.

```
public class Empresa {
    public static void listarFuncionarios(List<Funcionario> funcionarios) {
        for (Funcionario funcionario : funcionarios) {
            funcionario.receberSalario();
        }
    }

    public static void main(String[] args) {
        Gerente g1 = new Gerente(10, "Menino Ney");
        Vendedor v1 = new Vendedor(7, "Rainha Elizabeth");
        Vendedor v2 = new Vendedor(5);
        List<Funcionario> funcionarios = Arrays.asList(g1, v1, v2);
        listarFuncionarios(funcionarios);
    }
}
```

## Exercício

Crie uma hierarquia de classes conforme abaixo com os seguintes atributos e comportamentos (observe a tabela), utilize os seus conhecimentos e distribua as características de forma que tudo o que for comum a todos os animais fique na classe `Animal`:



| Cachorro | Cavalo | Preguiça |
| --- | --- | --- |
| Possui `nome` | Possui `nome` | Possui `nome` |
| Possui `idade` | Possui `idade` | Possui `idade` |
| Deve `emitirSom()` | Deve `emitirSom()` | Deve `emitirSom()` |
| Deve `correr()` | Deve `correr()` | Deve `subirEmArvores()` |