

# Programação Orientada a Objetos

## **Polimorfismo e Sobrescrita e Sobrecarga de Métodos**

Prof. Dirson Santos de Campos  
dirson\_campos@ufg.br

**(16/10/2023)**

# Tópicos

- **Polimorfismo**
  - **Casting**
  - **Operador instanceof**
- **Sobrescrita de Métodos e a anotação @Override**
- **Sobrecarga de Métodos**
- **Métodos protegidos Get e Set**

## Polimorfismo

# Programação Orientada a Objetos

## Polimorfismo (muitas formas)

- Etimologicamente a palavra vem do grego **Poly** (muito, numeroso, frequente) e **Morph** (forma).
- Em Orientação a Objetos (OO), permite que um mesmo objeto se manifestar de diferentes formas;
- Permite que uma mesma operação possa ser definida para diferentes tipos de classes, e cada uma delas a implementa como sua própria necessidade;

# Programação Orientada a Objetos

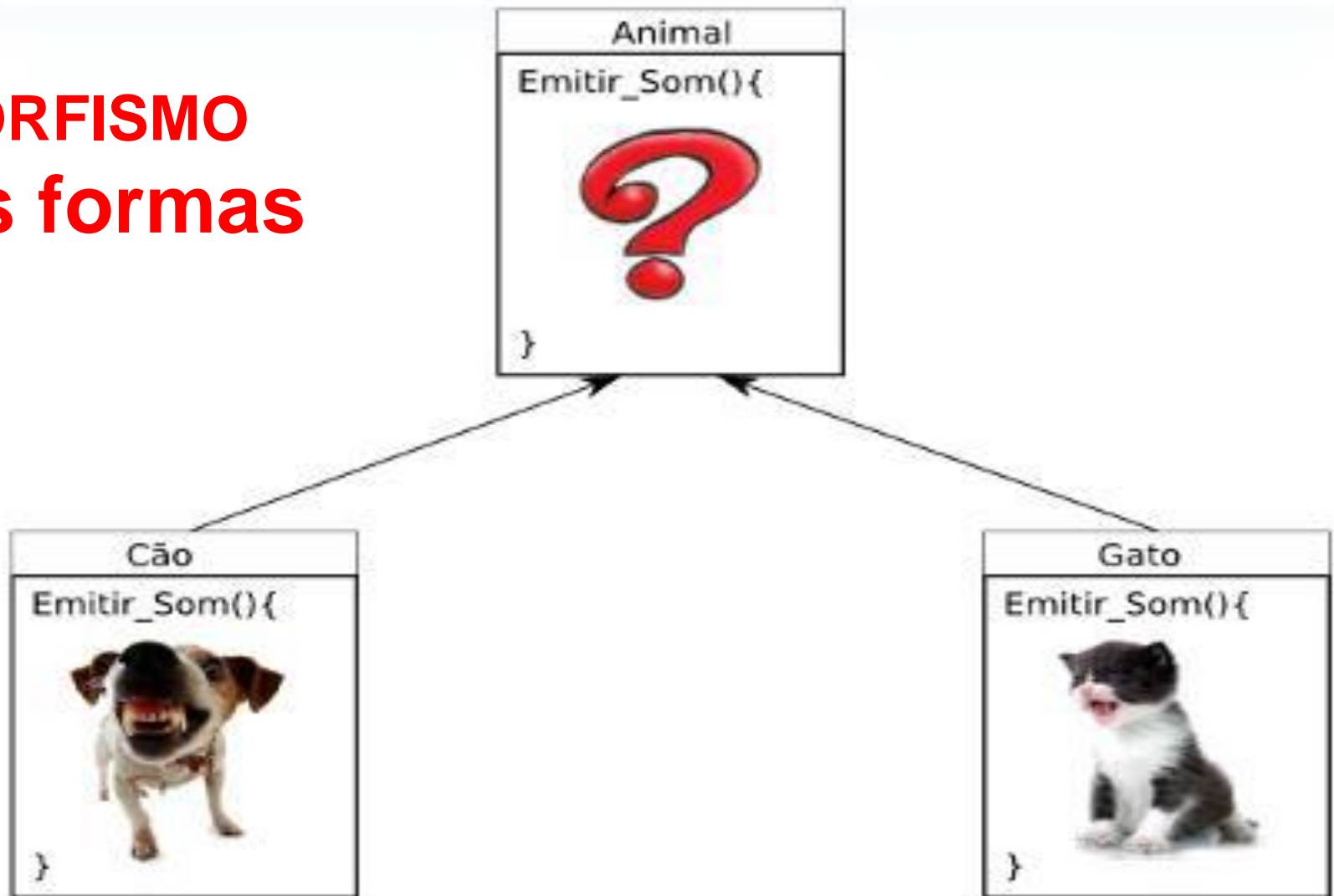
## Polimorfismo em OO

- É o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem se comportar de forma diferente;
- Invocar métodos que têm a mesma assinatura, mas com comportamentos distintos ... especializados para cada subclasse.
- O polimorfismo permite “programar no geral” em vez de “programar no específico”.

# Programação Orientada a Objetos

## Polimorfismo em Herança

**POLIMORFISMO**  
**Muitas formas**



# Programação Orientada a Objetos

## Analizando o Polimorfismo da figura do slide anterior

- A **superclasse Animal** tem um método **Emite\_som()**, mas sabe-se que um Objeto animal é capaz de emitir sons, mas na superclasse não foi definido que animal se refere.
- As **subclasses Cão e Gato** também tem cada uma o mesmo método **Emite\_Som()**, porém o som emitido pelo objeto cão é um (latido) e pelo objeto gato é outro (miado).

# Programação Orientada a Objetos

## Polimorfismo em Java na prática

```
public class Animal {  
    private String nome;  
    private int coordenadaX;  
    private int coordenadaY;  
  
    public Animal(String nome) {  
        this.nome = nome;  
    }  
  
    public Animal() {  
        this.nome = "anonimo";  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    protected void setCoordenadas(int x, int y) {  
        coordenadaX = x;  
        coordenadaY = y;  
    }  
  
    public void mover(int x, int y) {  
        System.out.println("Não sei me mover");  
    }  
}
```



# Programação Orientada a Objetos

## Polimorfismo em Java na prática

```
class Anfibio extends Animal {  
    public Anfibio(String nome) {  
        super(nome);  
    }  
    public void mover(int x, int y) {  
        setCoordenadas(x, y);  
        System.out.println("Movimento do Anfibio");  
    }  
}
```

```
class Ave extends Animal {  
    public Ave(String nome) {  
        super(nome);  
    }  
    public void mover(int x, int y) {  
        setCoordenadas(x, y);  
        System.out.println("Movimento da Ave");  
    }  
}
```

```
class Peixe extends Animal {  
    public Peixe(String nome) {  
        super(nome);  
    }  
    public void mover(int x, int y) {  
        setCoordenadas(x, y);  
        System.out.println("Movimento do Peixe");  
    }  
}
```

# Programação Orientada a Objetos

## Polimorfismo em Java na prática (cont..)

```
public class MundoAnimal {  
    public static void main (String args[]) {  
        Animal reino[];  
        reino = new Animal[3];  
  
        reino[0] = new Anfibio("Salamandra");  
        reino[1] = new Ave("Quero-quero");  
        reino[2] = new Peixe("Dourado");  
  
        for (int i=0; i<3; i++) {  
            reino[i].mover(10, 10);  
        }  
    }  
}
```

← Temos um vetor de Animais.

← Os elementos do vetor são instanciados com subclasses de animas.

← O método `mover()` é chamado para cada elemento do vetor.

• Qual método `mover()` vai ser chamado? O método da classe `Animal` ou os métodos especializados de cada uma das subclasses?

# Programação Orientada a Objetos

## Polimorfismo em prática (cont..)

- A decisão sobre qual método sobrescrito deve ser selecionado (dentro da hierarquia de classes) é tomada em tempo de execução, considerando a classe da instância (objeto) que o está chamando.
- O polimorfismo permite codificar programas que processam objetos que compartilham a mesma superclasse como se todos eles fossem objetos daquela superclasse (simplificando a programação).

# Programação Orientada a Objetos

## Polimorfismo em prática (cont..)

- Com o polimorfismo, podemos projetar e implementar sistemas facilmente extensíveis ... novas classes podem ser adicionadas com pouca ou nenhuma modificação nas partes gerais do programa ... desde que as novas classes façam parte da hierarquia de herança já existente.

Retângulo, círculo e triângulo são formas geométricas que podem ser desenhadas



Com o polimorfismo o método `desenhar()` de cada subclasse pode ser sobrescrito para que se comporte de maneira diferente.

Um programa que faz desenhos e que utiliza formas geométricas não precisa se preocupar em chamar métodos específicos de cada forma ... basta chamar o método `desenhar()` da hierarquia ... se a subclasse for um círculo ... seu método específico será chamado.



# Programação Orientada a Objetos

## ***Casting* ou Moldagem (tipos primitivos)**

- Utilizado para converter um objeto ou tipo primitivo de um tipo/classe para outro;
- Suponhamos a necessidade de armazenar um valor `int` dentro de um `double`... como a precisão de um `double` é maior ... a conversão é natural ...

```
int i = 3;
```

```
double pi= i + .14159;
```

- Quando há perda de precisão ... O *casting* é necessário ...

```
double pi= 3.14159;
```

```
int i = (int) pi; // Ao final i vale 3
```

# Programação Orientada a Objetos

## *Casting* de Objetos

- Em aplicações que exploram o polimorfismo é comum a necessidade de fazer com que um objeto se passe por outro.
- O *casting* não modifica o objeto, é o receptor do *cast* que constitui um novo objeto ou um novo tipo ...

• Ex:

```
Anfibio sapo = new Anfibio("Sapo Boi");
```

```
Animal animal = (Animal) sapo;
```

# Programação Orientada a Objetos

## Casting de Objetos

```
public class Teste {  
  
    public static void apresentar(Animal a) {  
        System.out.println( a.getNome() );  
        a.mover(10,10);  
    }  
  
    public static void main (String args[]) {  
  
        Object lista[];  
  
        lista = new Object[3];  
  
        lista[0] = new Anfibio("Salamandra");  
        lista[1] = new Ave("Quero-quero");  
        lista[2] = new Peixe("Dourado");  
  
        for (int i=0; i<3; i++) {  
            apresentar( (Animal) lista[i]);  
        }  
    }  
}
```

Temos um vetor de Objetos.

Um método que recebe um animal, mostra seu nome e o movimento

Antes de chamar o método, os objetos são transformados em Animais ...

Saída do código

```
Salamandra  
Movimento do Anfíbio  
  
Quero-quero  
Movimento da Ave  
  
Dourado  
Movimento do Peixe
```

# Programação Orientada a Objetos

## Exemplo: Herança e Polimorfismo

```
package PacoteHeranca1;
public class Funcionario {
    protected String nome;
    protected String cpf;
    protected double salario;

    public double bonificacao(){
        double b = salario * 0.10;
        return b;
    }
    public double getSalario() {
        return salario;
    }
    public void setSalario(double salario) {
        this.salario = salario;
    }
}
```



# Programação Orientada a Objetos

## Operador `instanceof`

- Utilizado para determinar o tipo de um objeto em tempo de execução;
- Utilizado em situações onde alguma operação específica de uma subclasse precisa ser chamada, mas antes é necessário verificar se o objeto que vai chamar o método é do tipo correto.
- Utilização: **`refObjeto instanceof nomeClasse`**

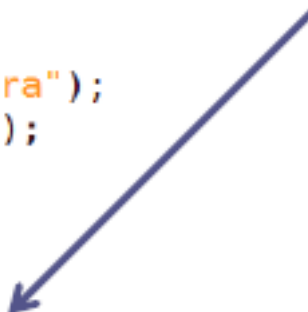
–Retorna um valor booleano (true ou false) indicando se o objeto referenciado (refObjeto) é realmente uma instância da classe (nomeClasse)

# Programação Orientada a Objetos

## Instanceof – Exemplo 1

```
public class MundoAnimal {  
    public static void main (String args[]) {  
        Animal reino[];  
        reino = new Animal[3];  
  
        reino[0] = new Anfibio("Salamandra");  
        reino[1] = new Ave("Quero-quero");  
        reino[2] = new Peixe("Dourado");  
  
        for (int i=0; i<3; i++) {  
            if (reino[i] instanceof Peixe) {  
                ( (Peixe) reino[i] ).nadar();  
            }  
            else {  
                reino[i].mover(10,10);  
            }  
        }  
    }  
}
```

Se a instância for da classe Peixe, então chama o método nadar (o *casting* é necessário para evitar um erro de compilação).



## Sobrescrita de Métodos

# Programação Orientada a Objetos

## Sobrescrita ou Sobreposição

- É a implementação de um método de uma superclasse na subclasse.
- A sobrescrita de métodos consiste basicamente em criar um novo método na classe filha **contendo a mesma assinatura e mesmo tipo de retorno do método sobrescrito.**

# Programação Orientada a Objetos

## Sobrescrita ou Sobreposição

- Em Java, a sobrescrita de métodos permite que uma subclasse forneça a sua própria implementação para um método herdado da sua superclasse.

# Programação Orientada a Objetos

## Anotação @Override

- É fortemente recomendável usar a anotação **@Override** em um método sobrescrito
- Facilita a leitura e compreensão do código
- Avisamos ao compilador (boa prática)

# Programação Orientada a Objetos

## Exemplo de Sobrescrita

```
public class Animal {  
    public String emitirSom() {  
        return "Som genérico de animal";  
    }  
}
```

```
public class gato extends Animal {  
    @Override  
    public String emitirSom() {  
        return "Miando";  
    }  
}
```

# Programação Orientada a Objetos

## Exemplo de Sobrescrita

- Neste exemplo, a classe gato ao sobrescreve o método emitirSom() da sua superclasse Animal.
- Quando chamamos o método emitirSom() na classe gato, ele retorna o som "Miando" em vez do som genérico "Som genérico de animal" da classe Animal.



## Sobrecarga de Métodos

# Programação Orientada a Objetos

## Sobrecarga de Métodos

- A sobrecarga de um método é uma situação em que o nome do método na subclasse é o mesmo que o da superclasse, mas há diferenças na lista de parâmetros ou no tipo de retorno.
- É importante observar que, tanto na sobrecarga quanto na sobrescrita, o nome do método deve ser idêntico na subclasse e na superclasse.

# Programação Orientada a Objetos

## Sobrecarga de Métodos

- Em Java, a sobrecarga ocorre mesmo na mesma classe, sem necessidade de haver herança.

# Programação Orientada a Objetos

## Exemplo de Sobrecarga

```
public class Calculadora{
```

```
//Somar dois valores (a e b)
```

```
public int somar(int a, int b){
```

```
return a + b;
```

```
}
```

```
//Desta vez, somamos três valores (a, b e c)
```

```
public int somar(int a, int b, int c){
```

```
return a + b + c;
```

```
}
```

```
}
```

# Programação Orientada a Objetos

## Exemplo de Sobrecarga

- Nesse exemplo, a classe Calculadora tem dois métodos chamados somar, mas eles diferem no número de parâmetros que cada um leva, embora o tipo de retorno no dois seja o mesmo, só a diferença no número de parâmetros é suficiente para ocorrer a sobrecarga.

# Programação Orientada a Objetos

## Exemplo de Sobrecarga

- Ao se chamar o método passarmos dois parâmetros **somar(1, 2)**, o primeiro método será chamado.
- Se ao se chamar o método passarmos três parâmetros **somar(1, 2, 3)**, o segundo método será chamado.

# Programação Orientada a Objetos

## Exemplo de Sobrecarga

- Quando um método é chamado com um determinado conjunto de argumentos, o compilador Java seleciona automaticamente o método apropriado com base na lista de argumentos fornecidos.

# Programação Orientada a Objetos

## Diferença entre Sobrescrita e Sobrecarga

- A principal diferença entre as duas é que, na **sobrescrita**, a lista de parâmetros e o tipo de retorno devem ser os mesmos da superclasse, enquanto, na **sobrecarga**, o tipo de retorno ou a lista de parâmetros devem ser diferentes.



## Métodos Get e Set

# Programação Orientada a Objetos

## Implementando métodos do tipo Get e Set

- ❑ A melhor forma que acessarmos os atributos de uma classe é utilizando métodos.
- ❑ Os métodos GET e SET são técnicas padronizadas para gerenciamento sobre o acesso dos atributos.
- ❑ Nesses métodos determinamos quando será alterado um atributo e o acesso ao mesmo, tornando o controle e modificações mais práticas e limpas, sem contudo precisar alterar assinatura do método usado para acesso ao atributo.

# Programação Orientada a Objetos

## Como criar métodos GET e SET usado em encapsulamento

- ❑ Na criação dos métodos para acesso a esses atributos privados devemos colocar GET ou SET antes do nome do atributo.
- ❑ Porém, existem diferenças entre os métodos, pois modularizamos um procedimento para método SET e uma função para método GET.

# Programação Orientada a Objetos

## Método get

- Quando formos acessar, “pegar” alguns atributos da classe, devemos utilizar os métodos GET.
- Esse método sempre retornará um valor, seja ele String, int ,double etc.
- Então devemos criar uma função.

# Programação Orientada a Objetos

**Exemplo de Método get não tem parâmetros de entrada (input) e o parâmetro de saída (output) é do mesmo tipo de dados da variável retornada.**

```
public Integer getNumero() {  
    return numero;  
}
```

# Programação Orientada a Objetos

## Método set

- ❑ Para alterarmos, modificarmos os valores de um atributo da classe de maneira protegida, utilizamos os métodos SET. Logo precisa de um parâmetro de entrada (*input*).

# Programação Orientada a Objetos

## Método set

- Esse método não terá um retorno (parâmetro de saída, ***output***), pois o atributo será somente modificado, criando um método de tipo VOID, sem retorno. Porém ele deve receber algum argumento para que possa ocorrer a devida alteração como *input*.
- Então devemos criar uma função.

# Programação Orientada a Objetos

**Método set para alterar um número.**

```
public void setNumero(Integer numero)
{
    this.numero = numero;
}
```