
Ufra
UNIVERSIDADE FEDERAL RURAL DA AMAZÔNIA

# Herança e Polimorfismo

Prof. MSc. Tiago Araújo

tiagodavi70@gmail.com

# Sumário

Herança de membros da classe

Encapsulamento

Polimorfismo

Palavra-chave super

# Herança

No mundo real, por meio da Genética, é possível herdarmos certas características de nossos ancestrais

- Atributos: cor dos olhos, cor da pele, doenças, etc.
- Comportamentos?

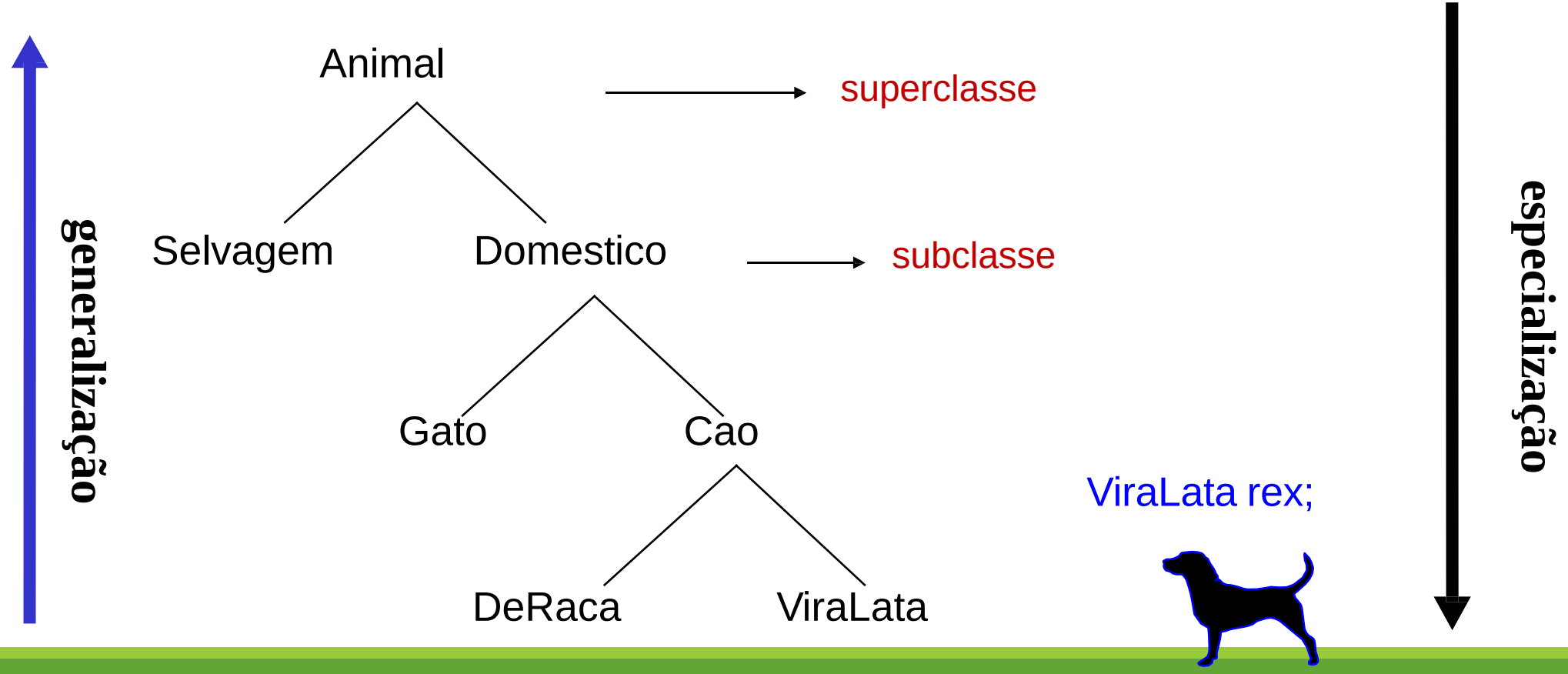
De forma similar, em POO as classes podem herdar

- Atributos (propriedades)
- Métodos (comportamento)

Chamamos este processo de herança

# Herança

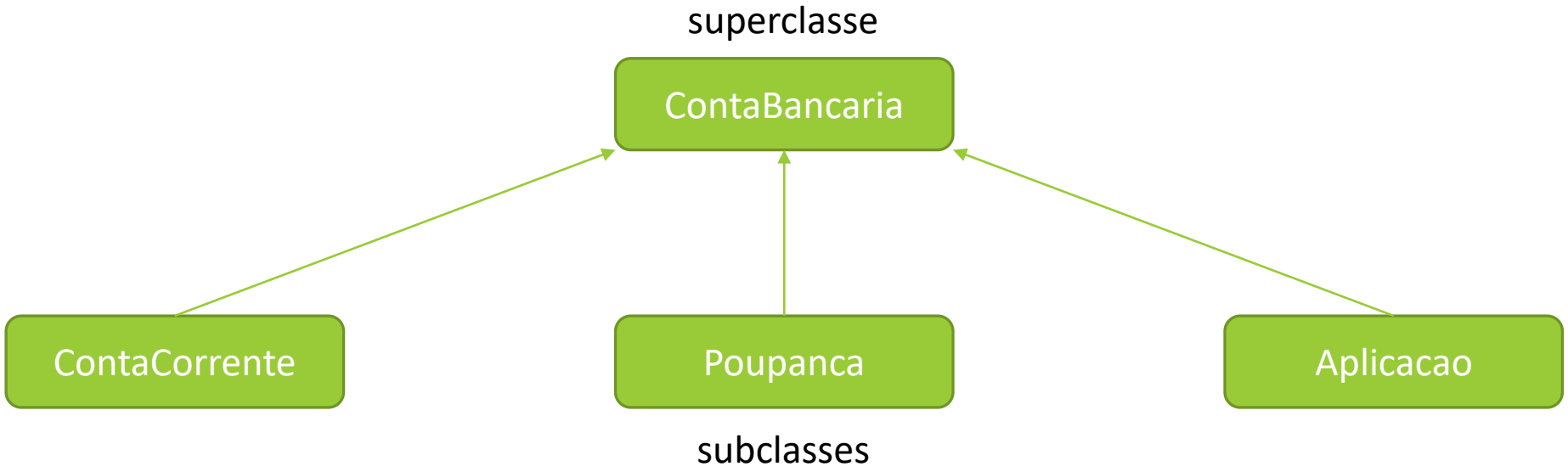
A herança pode ser repetida em cascata, criando várias gerações de classes



# Herança

**Classe mãe, superclasse, classe base**: A classe mais geral, a partir da qual outras classes herdam membros (atributos e métodos)

**Classe filha, subclasse, classe derivada**: A classe mais especializada, que herda os membros de uma classe mãe



# Herança

A herança é feita pela palavra-chave extends

```
public class Empregado {

}

public class Gerente extends Empregado {

}
```

# Herança

Herança permite a criação de classes com base em uma classe já existente

- Proporcionar o reuso de software
- Não é preciso escrever (e degubar) novamente
- Especialização de soluções genéricas já existentes

A ideia da herança é “ampliar” a funcionalidade de uma classe

Todo objeto da subclasse também é um objeto da superclasse, mas NÃO o contrário

# Encapsulamento

Subclasse herda todos os membros da superclasse

Construtores não são membros da classe

- Contudo, da subclasse é possível chamar um construtor da superclasse

Membros privados (-): ocultos na subclasse

- Acessíveis apenas por métodos

Membros protected (#): acessíveis na subclasse (e outras classes do mesmo pacote)

Membros package-private: acessíveis se a subclasse estiver no mesmo pacote da superclasse

Membros public (+): acessíveis na subclasse (e por qualquer outra classe)

Os membros herdados visíveis podem ser usados diretamente, como os membros da própria classe

# Encapsulamento

É possível declarar um campo na subclasse com o mesmo nome de um campo da superclasse

- Mesmo que os tipos sejam diferentes
- Ocultamento de campo (não recomendado)

É possível sobrescrever um método da superclasse, declarando um método com a mesma assinatura

- Polimorfismo

É possível declarar novos campos e métodos na subclasse

- Especialização

```
public class Bicicleta {

	public int cadencia;
	public int marcha;
	public int velocidade;

	public Bicicleta(int cadencia, int velocidade, int marcha) {
		this.marcha = marcha;
		this.cadencia = cadencia;
		this.velocidade = velocidade;
	}

	public int getCadencia() {
		return cadencia;
	}
	public void setCadencia(int novoValor) {
		 cadencia = novoValor;
	}
	public int getMarcha() {
		return marcha;
	}
	public void setMarcha(int novoValor) {
		marcha = novoValor;
	}
	public int getVelocidade() {
		return velocidade;
	}
	public void freiar(int decremento) {
		velocidade -= decremento;
	}
	public void acelerar(int incremento) {
		velocidade += incremento;
	}
}
```

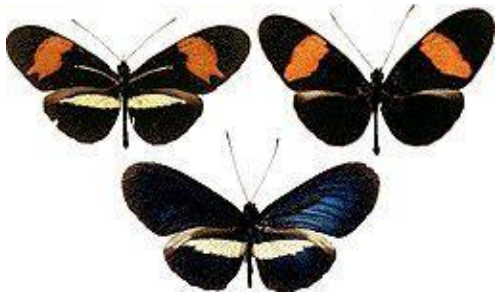
```
public class MountainBike extends Bicycle {

    // a subclasse MountainBike tem um campo a mais
    public int alturaBanco;

    // a subclasse MountainBike tem um construtor
    public MountainBike(int cadencia, int velocidade, int marcha, int alturaBanco) {
        super(startCadence, startSpeed, startGear);
        alturaBanco = startHeight;
    }

    // a subclasse MountainBike tem mais um métodos
    public void setAltura(int novoValor) {
        alturaBanco = novoValor;
    }
}
```

# Polimorfismo

Polimorfismo é um conceito que vem da biologia

É a capacidade de indivíduos ou organismos de uma mesma espécie apresentarem diferentes formas

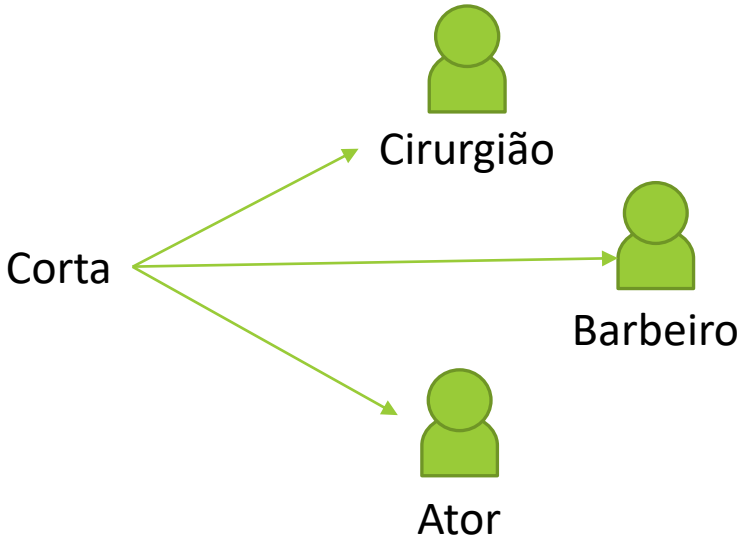
- Instâncias



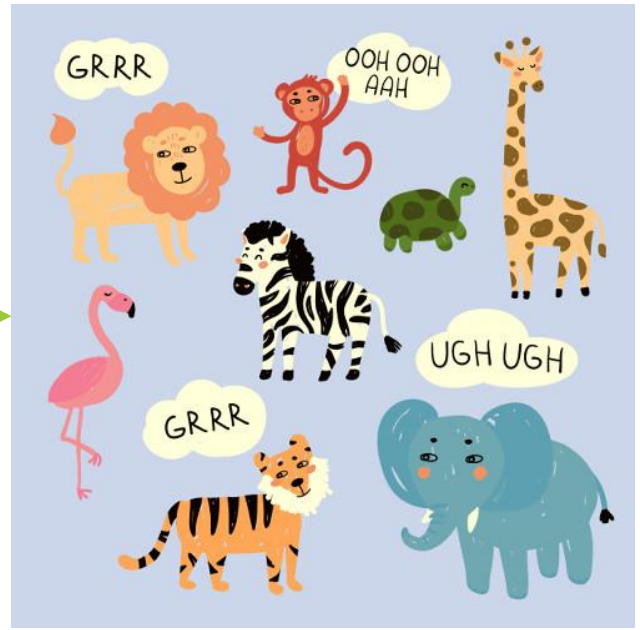
# Polimorfismo

Estendendo para POO

- Capacidade de objetos responderem a um MESMA MENSAGEM de maneira diferente
- Mensagem -> chamada de método
- Obtido através da sobreposição (reescrita) de métodos




Som



# Polimorfismo

Quando um método de um objeto é chamado, a JVM procura a implementação mais especializada

- Hierarquicamente, de baixo (especializado) para cima (geral)

Se o método não foi definido na classe derivada (subclasse), procura-se pela implementação da classe base (superclasse)

Quando o método é sobrescrito na subclasse, ele passa a ser o comportamento padrão daquela classe

- Mas ainda é possível acessar o método da superclasse

# Polimorfismo

A sobrescrita de métodos acontece quando um método da superclasse é redefinido na subclasse

- Mesma assinatura
- Mesmo tipo (ou subtipo) de retorno

Se quisermos aproveitar o comportamento definido pela superclasse, podemos chamar a implementação da superclasse

- Palavra-chave super
- Método da superclasse fica sobreposto (overriding)

```
public class Bicicleta {

    public int cadencia;
    public int marcha;
    public int velocidade;

    public Bicicleta(int cadencia, int velocidade, int marcha) {
        this.marcha = marcha;
        this.cadencia = cadencia;
        this.velocidade = velocidade;
    }

    // métodos, getters e setters....

    public descricao() {
        System.out.println("Bicicleta em " + "marcha " + this.marcha +
                           " com cadencia de " + this.cadencia + " RPM, e " +
                           " velocidade de " + this.velocidade + ". ");
    }
}
```

```
public class MountainBike extends Bicycle {

    // a subclasse MountainBike tem um campo a mais
    public int alturaBanco;

    // a subclasse MountainBike tem um construtor
    public MountainBike(int cadencia, int velocidade, int marcha, int alturaBanco) {
        super(cadencia, velocidade, marcha);
        this.alturaBanco = startHeight;
    }

    // Trocando a função
    public descricao() {
        System.out.println("Bicicleta em " + "marcha " + this.marcha +
                           " com cadencia de " + this.cadencia + " RPM, e " +
                           " velocidade de " + this.velocidade + ". " + "\n" +
                           "Com a altura de banco: " + this.alturaBanco);
    }
}
```

```
public class RoadBike extends Bicycle {

    public int larguraPneu;

    public RoadBike(int cadencia, int velocidade, int marcha, int larguraPneu) {
        super(cadencia, velocidade, marcha);
        this.larguraPneu = larguraPneu;
    }

    // métodos, getters e setters

    public descricao() {
        System.out.println("Bicicleta em " + "marcha " + this.marcha +
                           " com cadencia de " + this.cadencia + " RPM, e " +
                           " velocidade de " + this.velocidade + ". " + "\n" +
                           "Com a largura de pneu: " + this.larguraPneu);
    }

}
```

# Polimorfismo

A JVM chama o método correto de cada objeto, mesmo que eles estejam referenciado sob um tipo mais geral

Anotação @Override

- Em geral, é uma boa prática anotar os métodos que foram sobrescritos quando herdamos de uma superclasse
- Se o método não existe em nenhuma superclasse, o compilador acusa erro
- Também ajuda no entendimento do código

```
public class RoadBike extends Bicycle {

    public int larguraPneu;

    public RoadBike(int cadencia, int velocidade, int marcha, int larguraPneu) {
        super(cadencia, velocidade, marcha);
        this.larguraPneu = larguraPneu;
    }

    @Override
    public descricao() {
        System.out.println("Bicicleta em " + "marcha " + this.marcha +
                           " com cadencia de " + this.cadencia + " RPM, e " +
                           " velocidade de " + this.velocidade + ". " + "\n" +
                           "Com a largura de pneu: " + this.larguraPneu);
    }

}
```

# Polimorfismo

Alguns autores consideram a sobrecarga de métodos como uma forma de polimorfismo

Métodos com nomes iguais mas assinaturas diferentes

- Assinatura: nome do método, número de parâmetros e tipos dos parâmetros

Nome dos parâmetros e tipo de retorno NÃO fazem parte da assinatura

```
public int quadrado(int x) {
    return x * x;
}

public double quadrado(double x)
{
    return x * x;
}
```

# Palavra-chave *super*

Vimos anteriormente que a palavra chave *this* pode ser usada para referenciar o próprio objeto

Isso permite distinguir variáveis locais e campos do objeto que contém os mesmos nomes

A palavra-chave *super* tem uma função parecida em herança: acessar campos e métodos da superclasse

- Campos ocultos (hidden fields)
- Métodos sobrescritos (polimorfismo)
- Construtores da superclasse

# Palavra-chave *super*

Construtores

- Usamos super e o conjunto de argumentos entre parêntesis
- Deve ser sempre a primeira instrução do construtor da subclasse

Lembre-se que quando não há superclasse declarada, a superclasse direta é Object

```
public class Bicicleta {

     public int cadencia;
     public int marcha;
     public int velocidade;

     public Bicicleta(int cadencia,
                      int velocidade, int marcha) {
          this.marcha = marcha;
          this.cadencia = cadencia;
          this.velocidade = velocidade;
     }
}
```

```
public class RoadBike extends Bicycle {

     public int larguraPneu;

     public RoadBike(int cadencia, int velocidade,
                          int marcha, int larguraPneu) {
          super(cadencia, velocidade, marcha);
          this.larguraPneu = larguraPneu;
     }
}
```

# Outros

Métodos final não podem ser sobrescritos

- Em geral, para comportamentos que não devem ser mudados
- Quando os métodos são essenciais para manter a consistência dos objetos
- É recomendado que métodos chamados dentro de um construtor sejam final
  - Caso contrário, uma subclasse poderia alterar a maneira como o método é construído
  - Pode produzir resultados indesejados

A herança permite dizer que um objeto mais especializado também é um tipo mais geral

- Herança é uma relação “é um”
- Exemplo: MountainBike é uma Bicycle e um Object

Isso permite declarar tipos especializados como tipos mais gerais

- Casting implícito

# Outros

Se tentarmos associar um tipo mais geral a um tipo mais especializado, teremos um erro de compilação

Compilador não sabe que o tipo mais geral é também um tipo especializado

```
MountainBike mountainBike = new MountainBike();
Bicycle bike = new MountainBike();
Object obj = new MountainBike();

mountainBike = obj; // erro de compilção
```

Para corrigir isso, devemos fazer um casting explícito, informando ao compilador que o objeto é de fato um tipo mais especializado

Casting explícito só vale dentro da hierarquia de herança

```
MountainBike mountainBike = new MountainBike();
Bicycle bike = new MountainBike();
Object obj = new MountainBike();

mountainBike = (MountainBike)obj; // Ok
```

# Outros

Se durante a execução o objeto não for do tipo assinalado, uma exceção é lançada

Para evitar erro de execução, podemos testar o tipo da classe com *instanceof*

```
MountainBike mountainBike = new
MountainBike();
Bicycle bike = new MountainBike();
Object obj = new MountainBike();

if (obj instanceof MountainBike)
    mountainBike = (MountainBike)obj;
```

# Resumo

Herança de membros da classe

Encapsulamento

Polimorfismo

Palavra-chave super

Conversão de tipos (casting)