

Herança e Polimorfismo

Prof. Dr. Lucas C. Ribas

Disciplina: Programação Orientada a Objetos

Departamento de Ciências de Computação e Estatística



UNIVERSIDADE ESTADUAL PAULISTA
“JÚLIO DE MESQUITA FILHO”



IBILCE / UNESP - CÂMPUS DE SÃO JOSÉ DO RIO PRETO

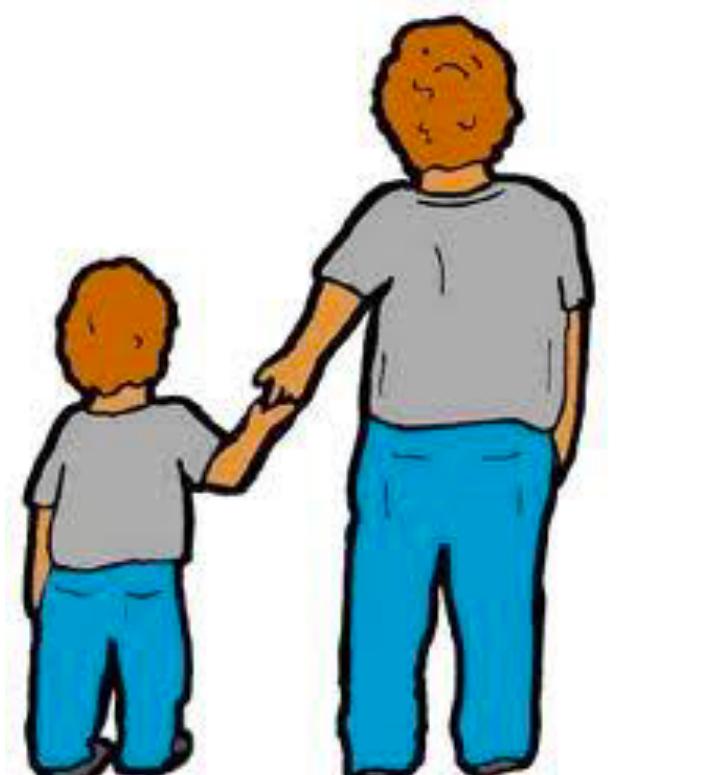


- Revisão de conceitos
- Herança de membros da classe
- Encapsulamento
- Polimorfismo
- Palavra-chave *super*
- Conversão de tipos (*casting*)

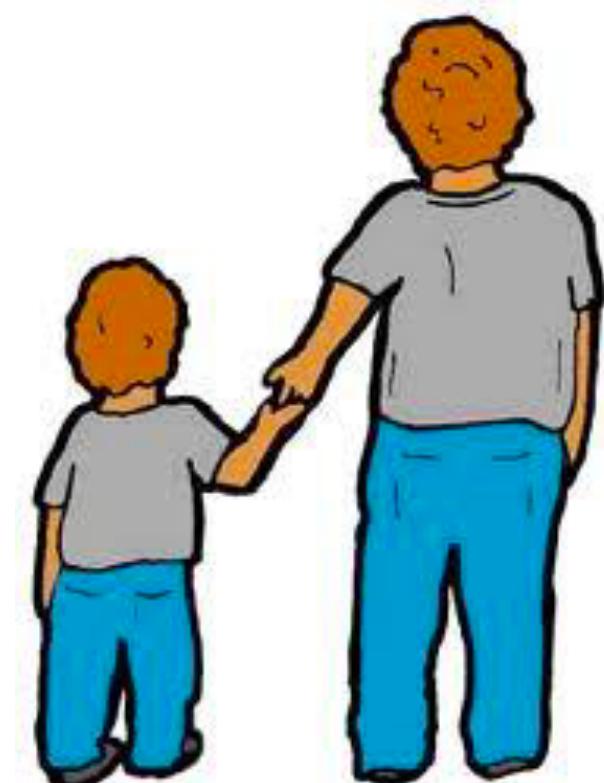
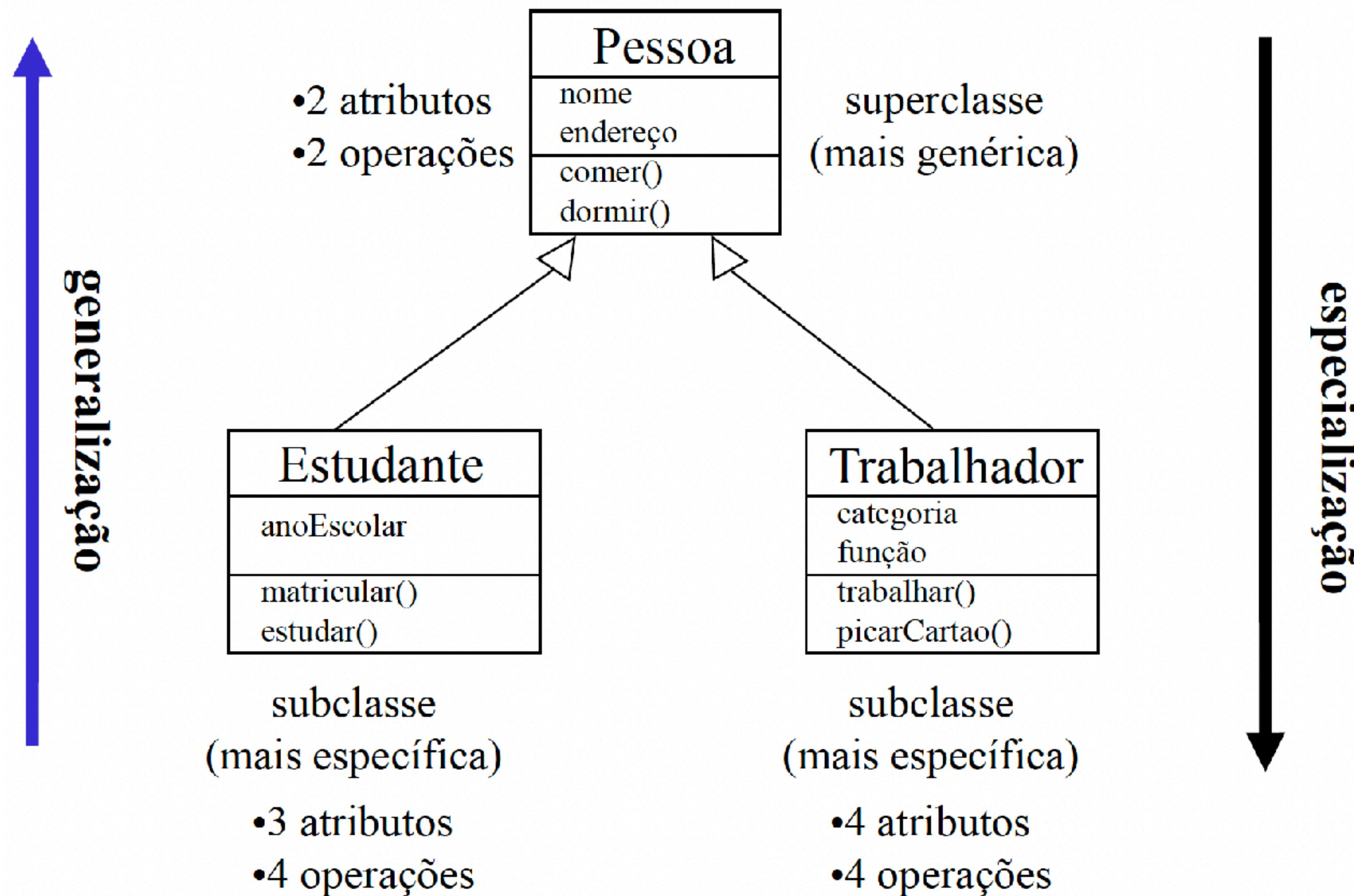


- No mundo real, por meio da Genética, é possível herdarmos certas características de nossos pais
 - Atributos: cor dos olhos, cor da pele, doenças, etc.
 - Comportamentos?
- De forma similar, em POO as classes podem herdar
 - Atributos (propriedades)
 - Métodos (comportamento)
- Chamamos este processo de **herança**

3

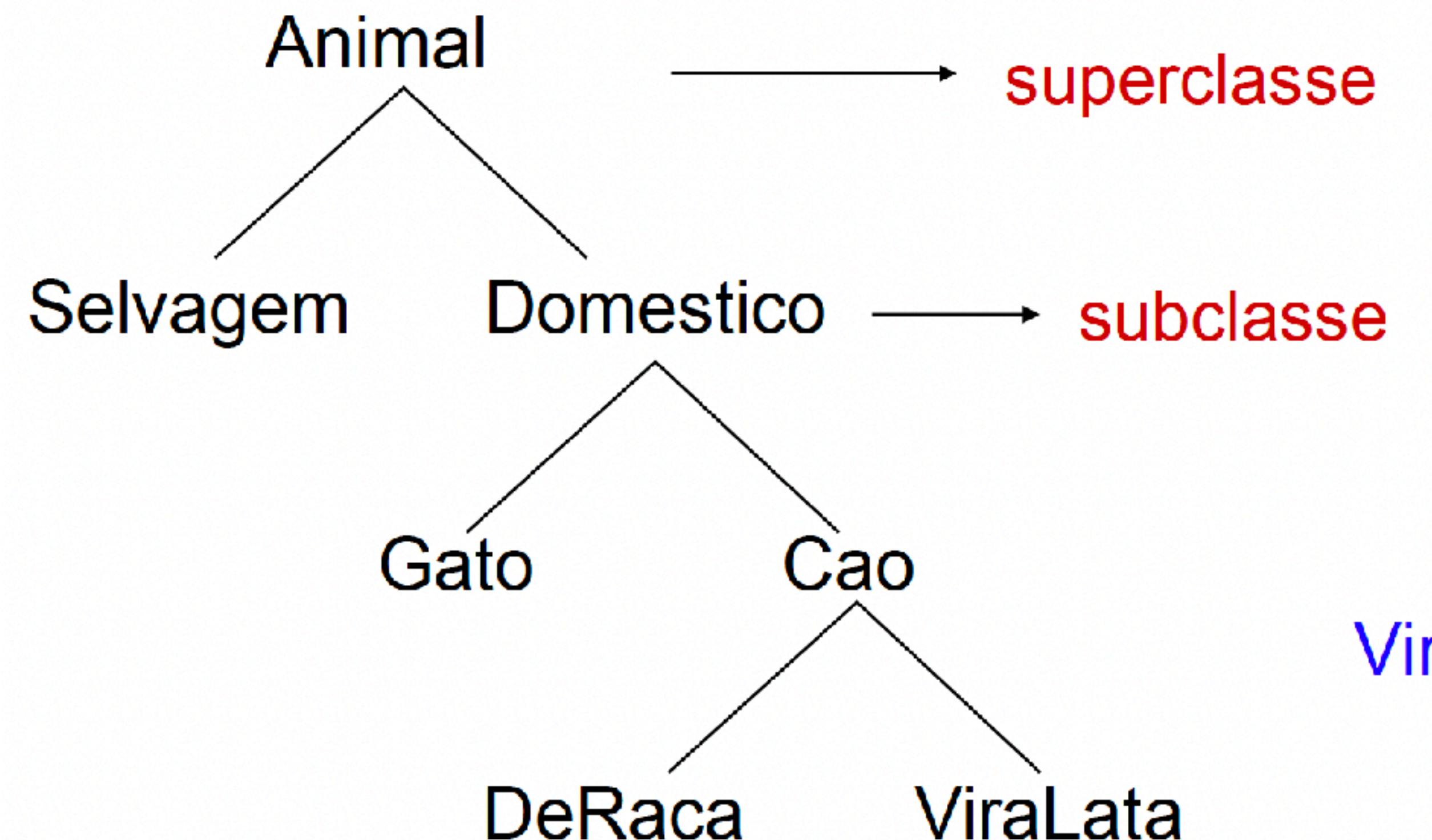


unesp

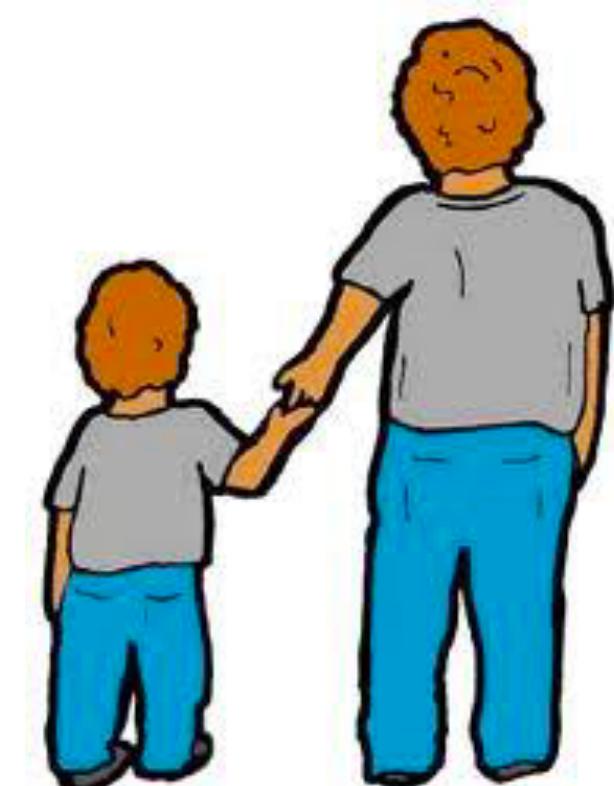
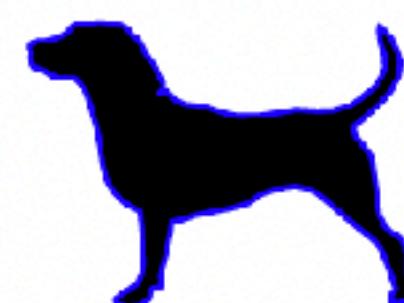




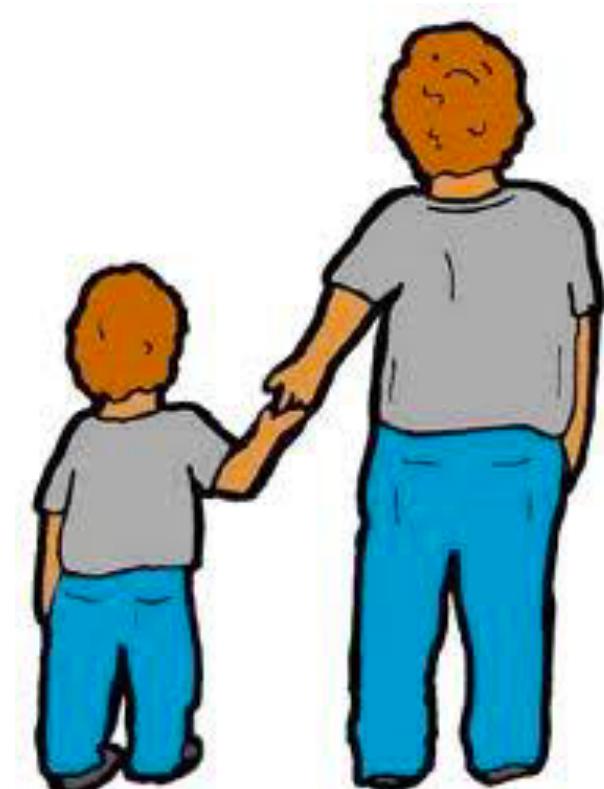
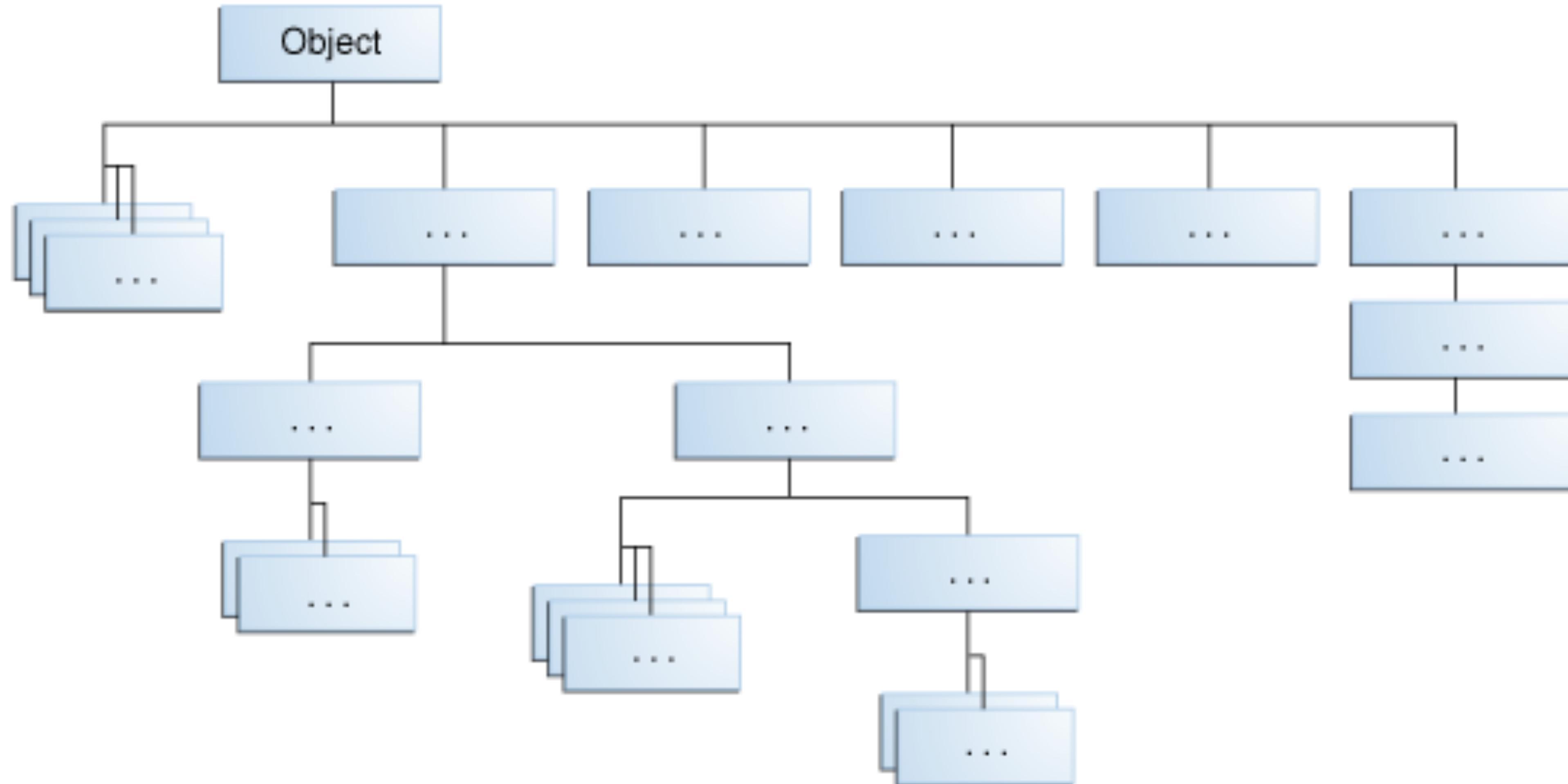
- A herança pode ser repetida em cascata, criando várias gerações de classes



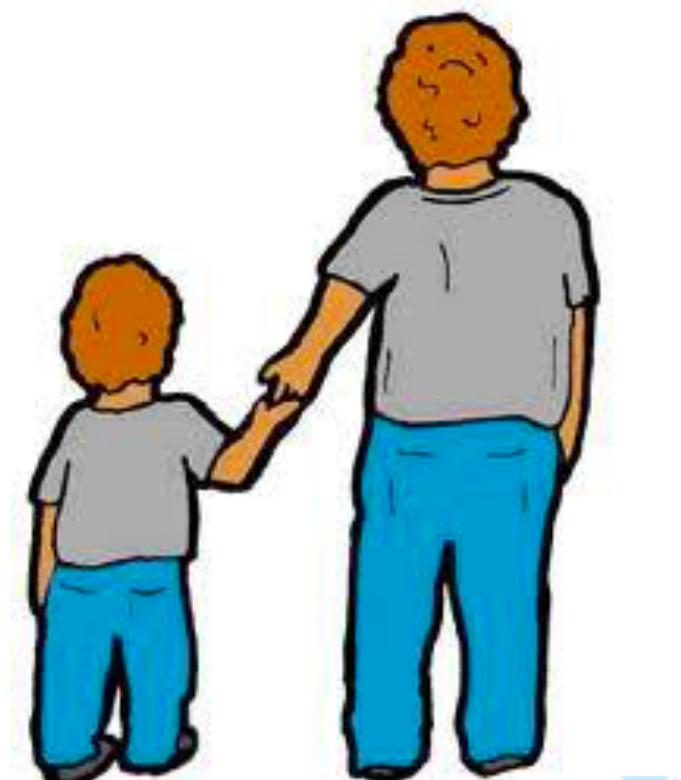
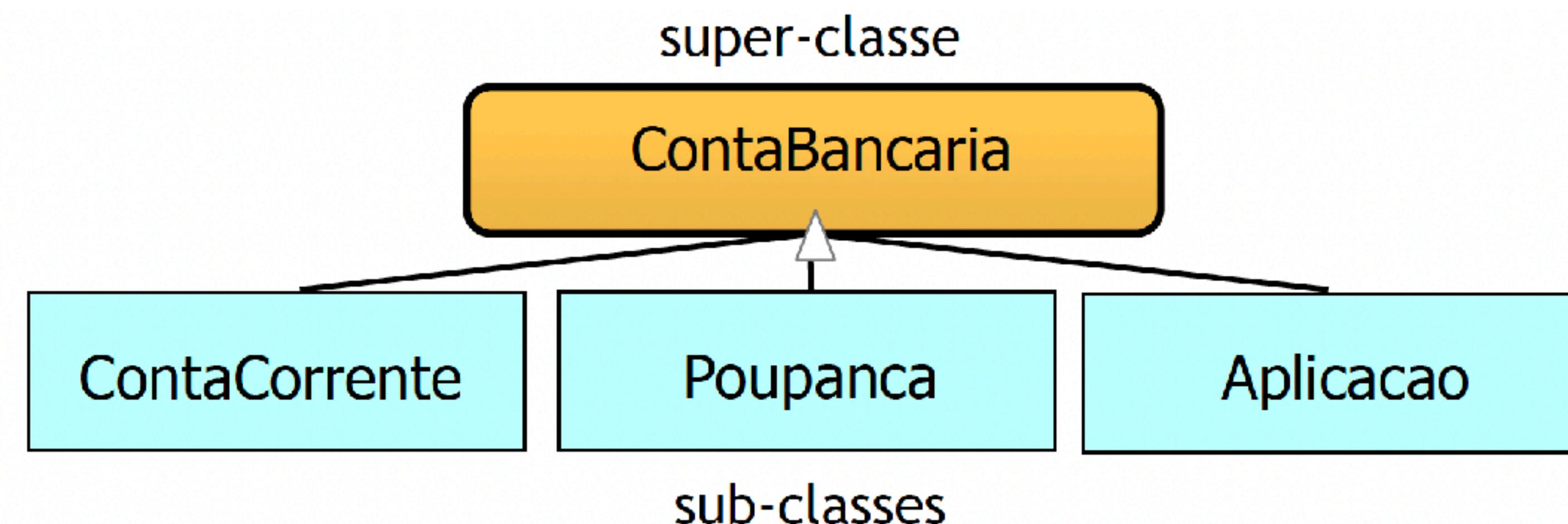
ViraLata rex;



Herança

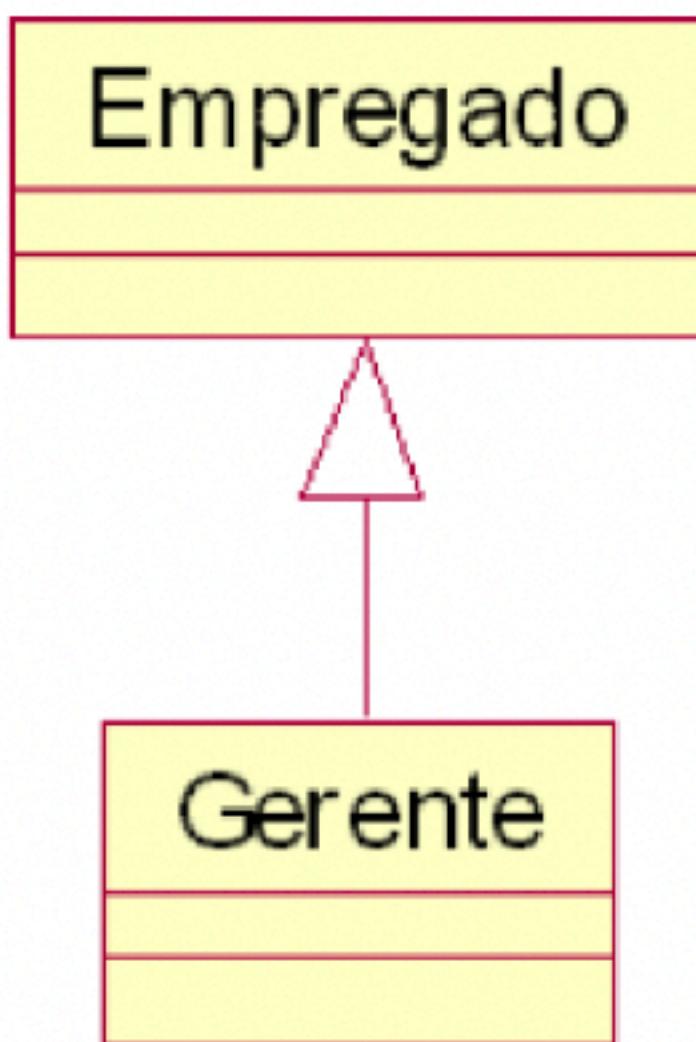


- **Classe mãe, superclasse, classe base:** A classe mais geral, a partir da qual outras classes herdam membros (atributos e métodos)
- **Classe filha, subclasse, classe derivada:** A classe mais especializada, que herda os membros de uma classe mãe

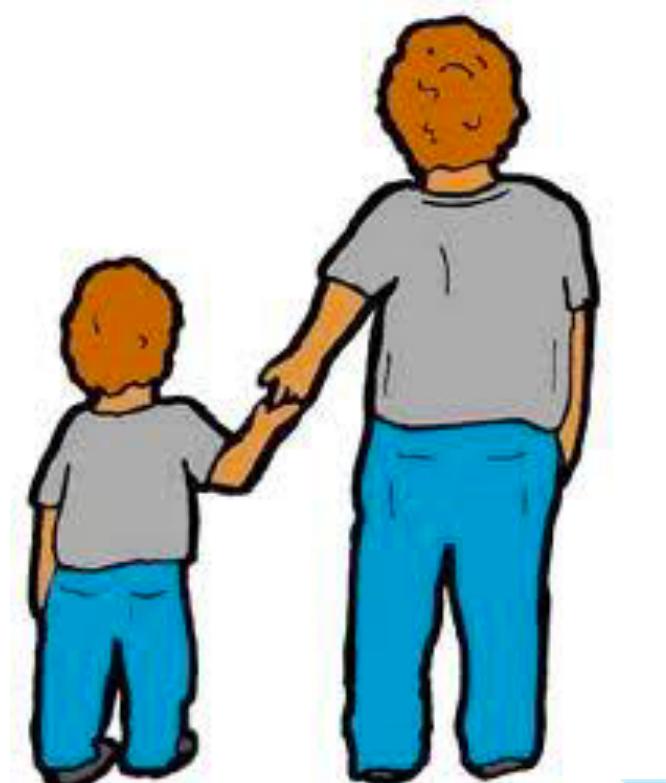




- A herança é feita pela palavra-chave **extends**

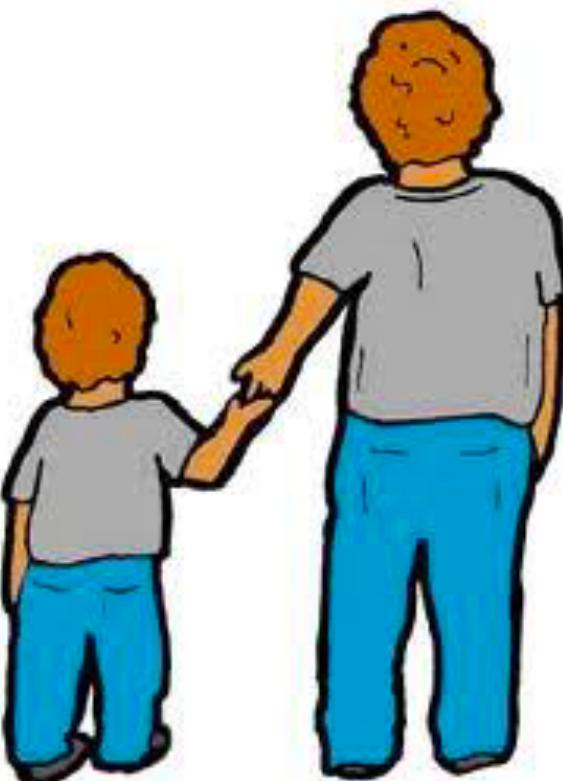


```
public class Empregado {  
}  
  
public class Gerente extends Empregado {  
}
```





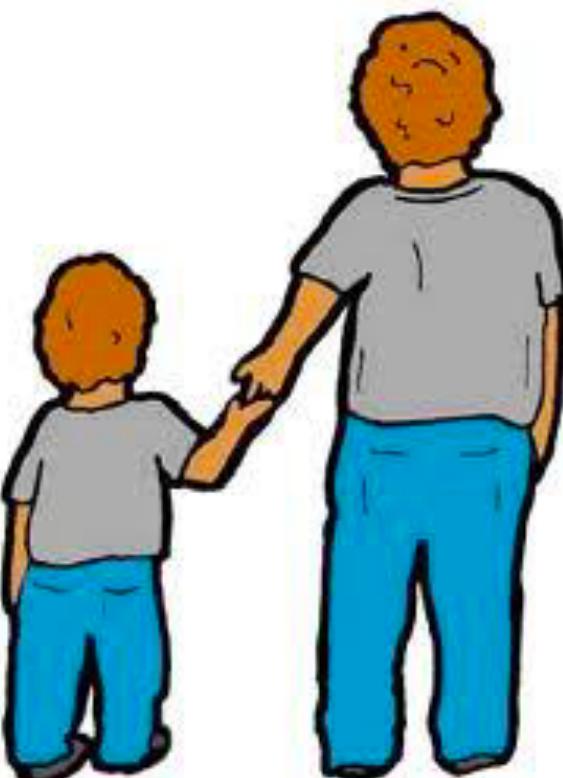
- Herança permite a criação de classes com base em uma classe já existente
 - Proporcionar o **reuso** de software
 - Não é preciso escrever (e degubar) novamente
 - Especialização de soluções genéricas já existentes
- A ideia da herança é “ampliar” a funcionalidade de uma classe
- Todo objeto da subclasse também **é um** objeto da superclasse, mas **NÃO** o contrário



Herança - Encapsulamento



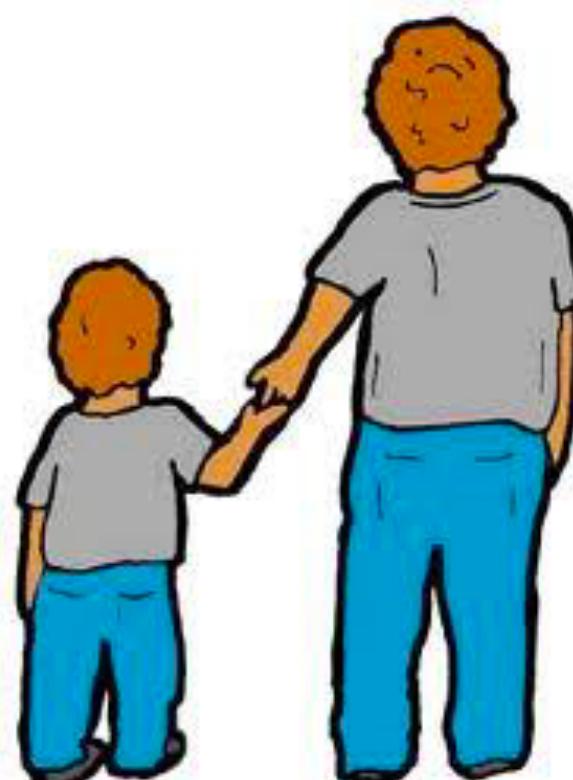
- Subclasse herda todos os membros da superclasse
 - Construtores não são membros da classe
 - Contudo, da subclasse é possível chamar um construtor da superclasse
 - Membros **privados (-)**: ocultos na subclasse
 - Acessíveis apenas por métodos
 - Membros **protected(#)**: acessíveis na subclasse (e 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



Herança - Encapsulamento



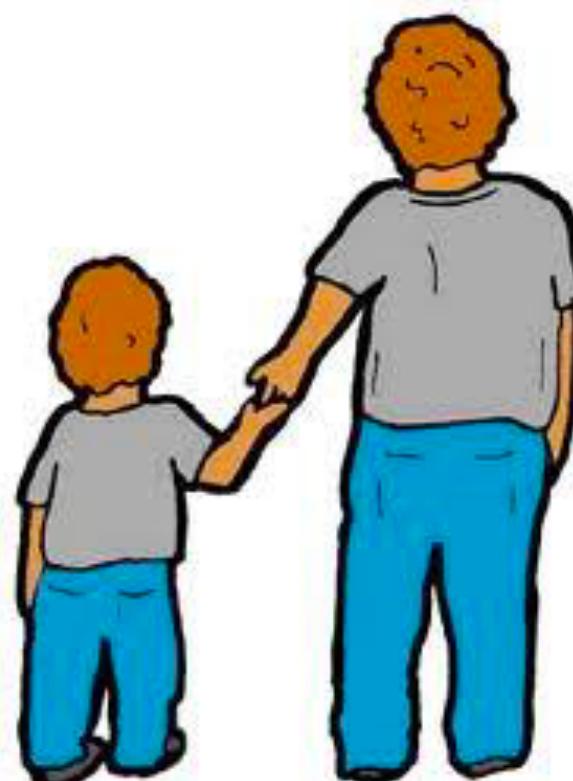
	Classe	Subclasse no mesmo pacote	Pacote (mesmo pacote)	Subclasse em outro pacote	Exterior (pacotes diferentes)
public	OK	OK	OK	OK	OK
protected	OK	OK	OK	OK	Não
<ausente>	OK	OK	OK	Não	Não
private	OK	Não	Não	Não	Não



Herança - Encapsulamento



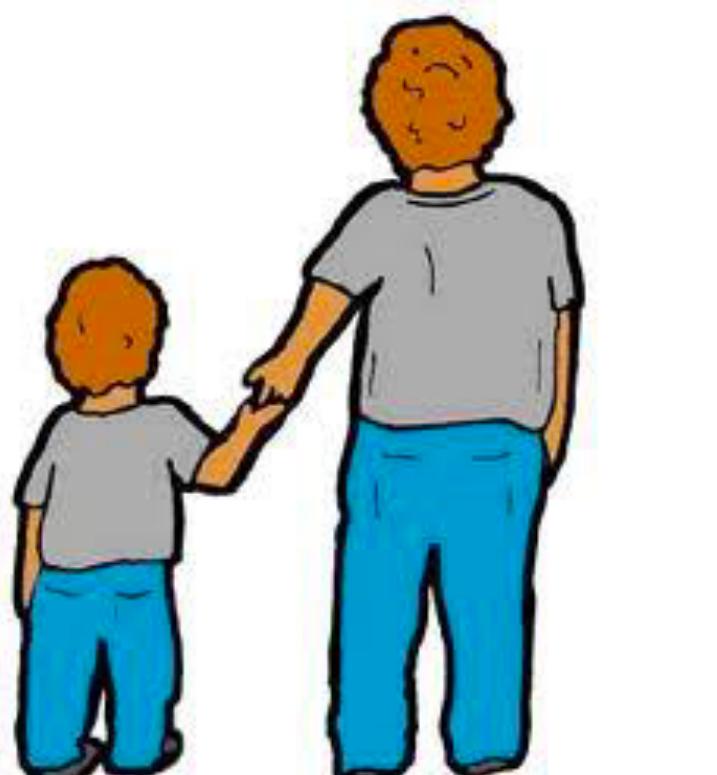
	Classe	Subclasse no mesmo pacote	Pacote (mesmo pacote)	Subclasse em outro pacote	Exterior (pacotes diferentes)
public			Pode tudo		
protected			Pode tudo no mesmo pacote, e em subclasses em qualquer lugar		
<ausente>			Pode tudo no mesmo pacote		
private			Não pode nada		



Herança - 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

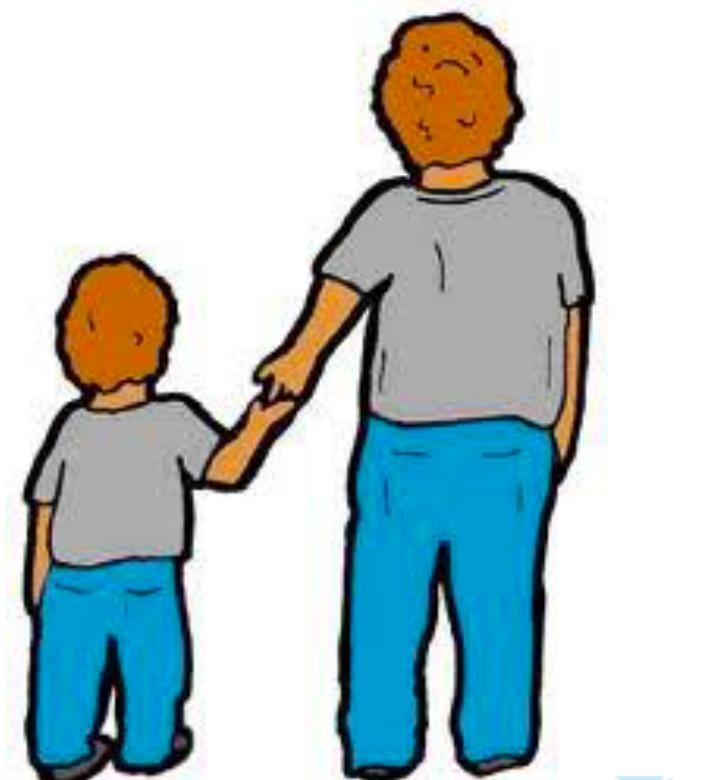


unesp

Herança - Encapsulamento



- Os membros herdados de uma classe podem ter seu acesso relaxados, mas não o contrário
- Por exemplo, um método **protected** na superclasse pode ser reescrito como **public** na subclasse, mas não como **private**



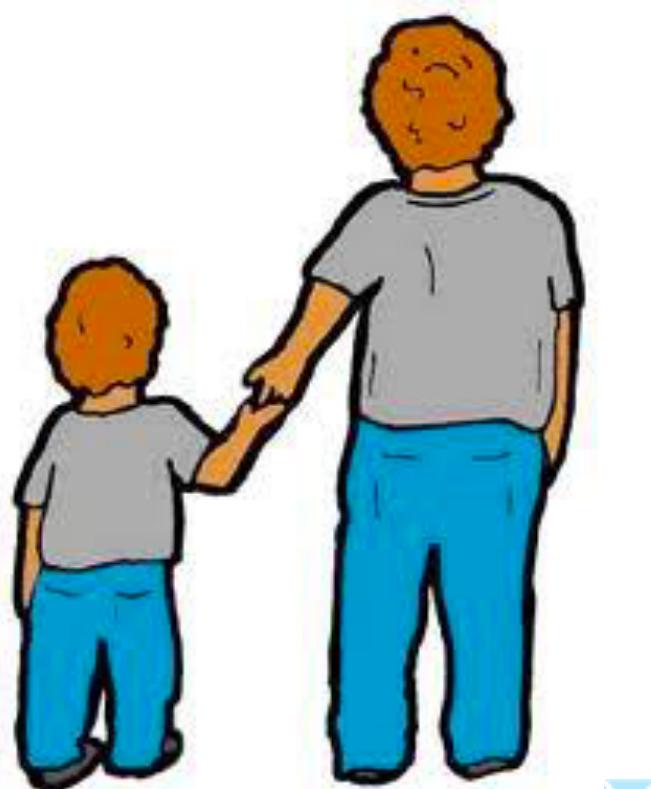
unesp 

The UNESP logo consists of a series of interlocking blue diamonds forming a larger diamond shape.

Herança



```
public class Bicicleta {  
    // a classe Bicicleta possui três campos  
    public int cadencia;  
    public int marcha;  
    public int velocidade;  
  
    // a classe Bicicleta possui um construtor  
    public Bicicleta(int cadenciaInicial, int velocidadeInicial, int marchaInicial) {  
        marcha = marchaInicial;  
        cadencia = cadenciaInicial;  
        velocidade = velocidadeInicial;  
    }  
  
    // a classe Bicicleta possui quatro métodos  
    public void setCadencia(int novoValor) {  
        cadencia = novoValor;  
    }  
  
    public void setMarcha(int novoValor) {  
        marcha = novoValor;  
    }  
  
    public void aplicarFreio(int decremento) {  
        velocidade -= decremento;  
    }  
  
    public void acelerar(int incremento) {  
        velocidade += incremento;  
    }  
}
```



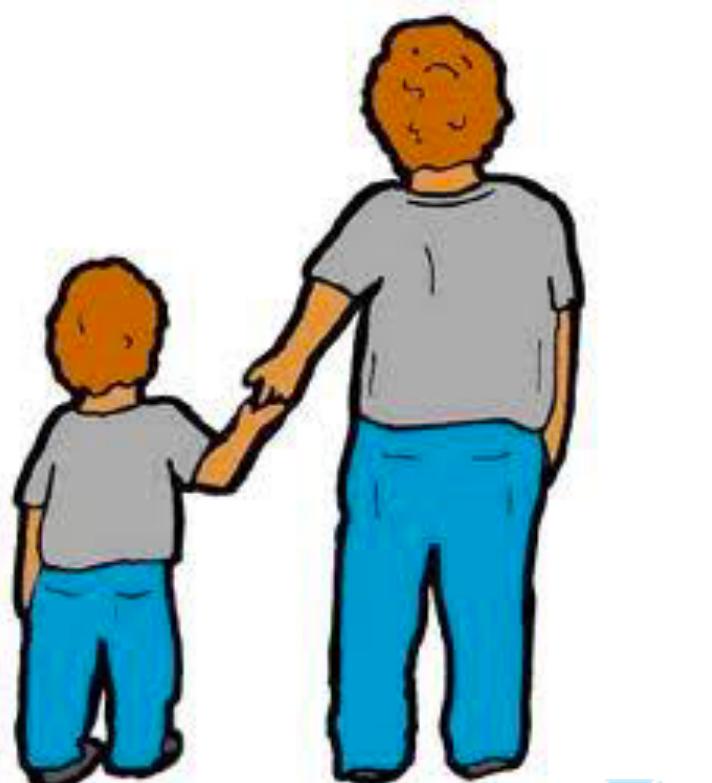
unesp



```
public class BicicletaMontanha extends Bicicleta {  
    // a subclasse BicicletaMontanha adiciona um campo  
    public int alturaDoSelim;  
  
    // a subclasse BicicletaMontanha possui um construtor  
    public BicicletaMontanha(int alturaInicial, int cadenciaInicial, int velocidadeInicial, int marchaInicial) {  
        super(cadenciaInicial, velocidadeInicial, marchaInicial);  
        alturaDoSelim = alturaInicial;  
    }  
  
    // a subclasse BicicletaMontanha adiciona um método  
    public void setAltura(int novoValor) {  
        alturaDoSelim = novoValor;  
    }  
}
```



- Se os campos da classe *Bicycle* fossem privados, subclasses não teriam acesso a eles
 - O acesso poderia acontecer indiretamente, através de métodos públicos ou protegidos da superclasse
- Uma forma de torná-lo acessíveis diretamente na subclasses, mas ainda com restrições, é usar modificador protegido
 - Subclasses e classes do mesmo pacote
- O ideal é manter sempre o mais escondido o possível



Polimorfismo

- Como já discutido, polimorfismo vem da biologia
- É a capacidade de indivíduos ou organismos de uma mesma espécie apresentarem diferentes formas
 - Instâncias



● 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





- 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



- 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*)



○ Exemplos

```
public class Bicicleta {  
    // campos  
    private int cadencia;  
    private int marcha;  
    private int velocidade;  
  
    // métodos  
    // getters e setters  
  
    public void imprimirDescricao() {  
        System.out.println("\nA bicicleta está na marcha " +  
            this.marcha + " com uma cadência de " +  
            this.cadencia + " e viajando a uma velocidade de " +  
            this.velocidade + ".");  
    }  
}
```

Polimorfismo



```
public class BicicletaMontanha extends Bicicleta {  
    private String suspensao;  
  
    public BicicletaMontanha(int cadenciaInicial, int velocidadeInicial,  
                            int marchaInicial, String tipoSuspensao) {  
        super(cadenciaInicial, velocidadeInicial, marchaInicial);  
        this.setSuspensao(tipoSuspensao);  
    }  
  
    public String getSuspensao() {  
        return this.suspensao;  
    }  
  
    public void setSuspensao(String tipoSuspensao) {  
        this.suspensao = tipoSuspensao;  
    }  
  
    public void imprimirDescricao() {  
        super.imprimirDescricao();  
        System.out.println("A " + "BicicletaMontanha tem uma " +  
                           getSuspensao() + " suspensão.");  
    }  
}
```

Polimorfismo



```
public class BicicletaEstrada extends Bicicleta {  
    private int larguraPneu;  
  
    public BicicletaEstrada(int cadenciaInicial, int velocidadeInicial,  
                           int marchaInicial, int novaLarguraPneu) {  
        super(cadenciaInicial, velocidadeInicial, marchaInicial);  
        this.setLarguraPneu(novaLarguraPneu);  
    }  
  
    public int getLarguraPneu() {  
        return this.larguraPneu;  
    }  
  
    public void setLarguraPneu(int novaLarguraPneu) {  
        this.larguraPneu = novaLarguraPneu;  
    }  
  
    public void imprimirDescricao() {  
        super.imprimirDescricao();  
        System.out.println("A BicicletaEstrada" +  
                           "possui pneus com " + getLarguraPneu() + " MM de largura.");  
    }  
}
```

Polimorfismo



```
public class TestarBicicletas {  
    public static void main(String[] args) {  
        Bicicleta bike01, bike02, bike03;  
        bike01 = new Bicicleta(20, 10, 1);  
        bike02 = new BicicletaMontanha(20, 10, 5, "Dupla");  
        bike03 = new BicicletaEstrada(40, 20, 8, 23);  
  
        bike01.imprimirDescricao();  
        bike02.imprimirDescricao();  
        bike03.imprimirDescricao();  
    }  
}
```



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

A bicicleta está na marcha 1 com uma cadênciade 20 e viajando a uma velocidade de 10.

A bicicleta está na marcha 5 com uma cadênciade 20 e viajando a uma velocidade de 10.
A BicicletaMontanha tem uma suspensão dupla.

A bicicleta está na marcha 8 com uma cadênciade 40 e viajando a uma velocidade de 20.
A BicicletaEstrada possui pneus com 23 MM de largura.

● 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
- Garante que você não cometa erros, como digitar incorretamente o nome do método ou usar uma assinatura diferente
- Também ajuda no entendimento do código

```
public class BicicletaEstrada extends Bicicleta {  
    ...  
    @Override  
    public void imprimirDescricao() {  
        // o código vai aqui  
    }  
}
```



- 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



○ Campos ocultos

- Se a subclasse tem um campo com o mesmo nome de um campo na superclasse
- Para acessar o campo da superclasse, deve-se usar o **super**
- Desaconselhável, pois torna o código de difícil interpretação



● Métodos sobrescritos

```
public class Superclass {  
    public void printMethod() {  
        System.out.println("Printed in Superclass.");  
    }  
}
```

```
public class Subclass extends Superclass {  
  
    @Override  
    public void printMethod() {  
        super.printMethod();  
        System.out.println("Printed in Subclass");  
    }  
  
    public static void main(String[] args) {  
        Subclass s = new Subclass();  
        s.printMethod();  
    }  
}
```

Palavra-chave *super*



● Métodos sobrescritos

- Compilando e executando Subclass

Printed in Superclass.
Printed in Subclass



● Construtores

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

```
public class BicicletaMontanha extends Bicicleta {  
    private String suspensao;  
  
    public BicicletaMontanha(int cadenciaInicial, int velocidadeInicial,  
                            int marchaInicial, String tipoSuspensao) {  
        super(cadenciaInicial, velocidadeInicial, marchaInicial);  
        this.setSuspensao(tipoSuspensao);  
    }  
}
```



○ Construtores

- Quando não há uma chamada explícita a um construtor da superclasse, o compilador Java insere uma chamada ao construtor sem argumentos da superclasse
- Se a superclasse não tem tal construtor, um erro é gerado

○ Importante

- Lembre-se que quando não há superclasse declarada, a superclasse direta é **Object**
- Quando nenhum construtor é definido na classe, o compilador Java cria um construtor padrão



- 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

```
class AlgoritmoXadrez {  
    enum JogadorXadrez { BRANCO, PRETO }  
    ...  
  
    final JogadorXadrez getPrimeiroJogador() {  
        return JogadorXadrez.BRANCO;  
    }  
    ...  
}
```



- 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

Classes e Métodos *final*



- Classes **final** não podem ser herdadas
 - Exemplo: classe String
 - Garante que a String não será manipulada

Conversão de Tipos (*Casting*)



- Como vimos, a herança permite dizer que um objeto mais especializado também é um tipo mais geral
 - Herança é uma relação “é um”
 - Exemplo: **BicicletaMontanha** é uma **Bicicleta** e um **Object**
- Isso permite declarar tipos especializados como tipos mais gerais
 - **Casting implícito**

```
BicicletaMontanha mountainBike = new BicicletaMontanha();  
Bicicleta bike = new BicicletaMontanha();  
Object obj = new BicicletaMontanha();
```

Conversão de Tipos (*Casting*)



- 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

```
BicicletaMontanha mountainBike = new BicicletaMontanha();  
Bicicleta bike = new BicicletaMontanha();  
Object obj = new BicicletaMontanha();  
  
mountainBike = obj; //erro no compilador
```

Conversão de Tipos (*Casting*)



- Para corrigir isso, devem o 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

```
BicicletaMontanha mountainBike = new BicicletaMontanha();  
Bicicleta bike = new BicicletaMontanha();  
Object obj = new BicicletaMontanha();  
  
mountainBike = (BicicletaMontanha) obj; //ok
```

Conversão de Tipos (*Casting*)



- 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`

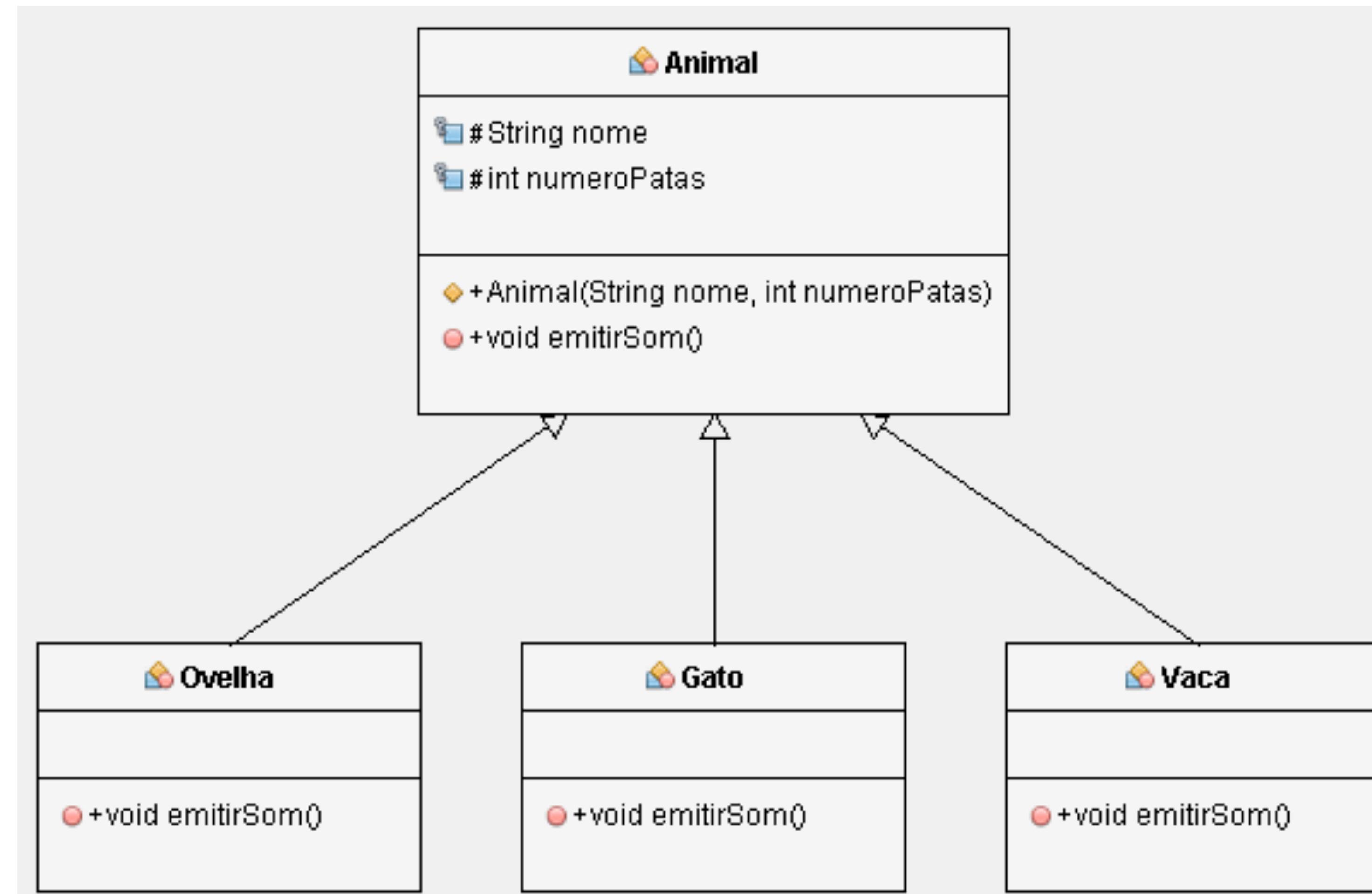
```
BicicletaMontanha mountainBike = new BicicletaMontanha();
Bicicleta bike = new BicicletaMontanha();
Object obj = new BicicletaMontanha();

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



- Revisão de conceitos
- Herança de membros da classe
- Encapsulamento
- Polimorfismo
- Palavra-chave *super*
- Conversão de tipos (*casting*)

● Implemente as classes representadas no diagrama



Exercício



```
public class Zoo {  
  
    public static void main(String[] args) {  
        Animal animal = new Animal("Bicho", 8);  
        Vaca vaca = new Vaca("Mimosa", 4);  
        Gato gato = new Gato("Garfield", 4);  
        Ovelha ovelha = new Ovelha("Dolly", 4);  
  
        Animal bichos[] = {animal, vaca, gato, ovelha};  
  
        // Aqui, cada instancia eh de um tipo especializado  
        if(vaca instanceof Animal)  
            System.out.println("vaca eh Animal");  
        if(gato instanceof Animal)  
            System.out.println("gato eh Animal");  
        if(ovelha instanceof Animal)  
            System.out.println("ovelha eh Animal");  
  
        System.out.println("-----\n");  
        for(int i=0 ; i < bichos.length ; i++) {  
            System.out.print(bichos[i].nome);  
  
            if(bichos[i] instanceof Vaca)  
                System.out.print(" eh uma vaca");  
        }  
    }  
}
```

```
        if(bichos[i] instanceof Gato)  
            System.out.print(" eh um gato");  
  
        if(bichos[i] instanceof Ovelha)  
            System.out.print(" eh uma ovelha");  
  
        System.out.print(", tem " + bichos[i].numeroPatas +  
                        " patas e emite o som: ");  
        bichos[i].emitirSom();  
        System.out.println();  
  
        // Aqui, bichos[] eh do tipo Animal  
        if(bichos[i] instanceof Animal)  
            System.out.println(bichos[i].nome +  
                            " eh um Animal.");  
  
        System.out.println();  
    }  
}
```



- Defina como deve ser o construtor de cada subclasse
- Utilize a classe Zoo para testar seu código
 - O que acontece se alguma classe não reescrever o método **emitirSom()** ?
- Defina um atributo a mais em cada subclasse, que seja único para cada tipo
 - Cor da lã (ovelha), corte ou leite (vaca), etc...
- Crie o método **toString()** em todas as classes, que descreva o objeto como um todo
 - Todos os atributos
- Modifique a classe Zoo para chamar esse método



- Altere os modificadores de acesso dos campos da classe Animal para private
- O que precisa ser alterado nas classes Gato, Vaca e Ovelha?
- O que precisa ser alterado na classe Zoo para que ela continue oferecendo a mesma funcionalidade?



- DEITEL, H. M. & DEITEL, P.J. "Java : como programar", Bookman, 2017.
- Material baseado nos slides:
 - Luiz E. Virgilio da Silva. Notas de Aula de Programação Orientada a Objetos (ICMC/USP).