



CCM310

Arquitetura de Software e

Programação Orientada a Objetos

Profa. Dra. Gabriela Biondi

Prof. Dr. Isaac Jesus

Prof. Dr. Luciano Rossi

Sobrecarga de Métodos

Sobrecarga de Métodos

- **Métodos Sobrecarregados:** são métodos com o mesmo nome porém com **assinaturas diferentes**
- A sobrecarga de métodos acontece com métodos **pertencentes à mesma classe**
- Já vimos algo semelhante na **Sobrecarga de Construtores**

- Exemplo sem Orientação a Objetos
- Sobrecarga de Função
- A função *calcula()* é sobrecarregada

```
1 #include<iostream>
2
3 int calcula (int x, int y)
4 {
5     return x + y;
6 }
7
8 float calcula (float x, float y)
9 {
10    return x + y;
11 }
12
13
14 int main()
15 {
16
17     int a = 5, b = 8;
18     float c = 9.5, d = 1.3;
19
20     std::cout << calcula(a,b) << std::endl;
21     std::cout << calcula(c,d) << std::endl;
22     //std::cout << calcula(a,c) << std::endl;
23 }
```

Exercício 1 - Sobrecarga de Métodos

- Crie uma classe, em Java, para calcular o quadrado de um número
- A classe deve conter o método *square()* que aceita parâmetro inteiro ou double

Polimorfismo

Polimorfismo

É um dos pilares da Orientação a Objetos

Pilares da Orientação a Objetos

Programação Orientada a Objetos

Abstração

Encapsulamento

Herança

Polimorfismo

Polimorfismo: Traduzindo do grego, significa “*muitas formas*”.

Conceitos de Polimorfismo

- Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas
- O polimorfismo está ligado aos conceitos de Herança e Hierarquia de Classes

Conceitos de Polimorfismo

□ Exemplo:



Podemos nos referir a um **Gerente** como sendo um **Funcionário**:

```
Funcionario func;  
func = new Gerente();
```

- O objeto *func* é declarado como um **Funcionário**, mas é instanciado como um **Gerente**!
- *func* vai agir como um Gerente!

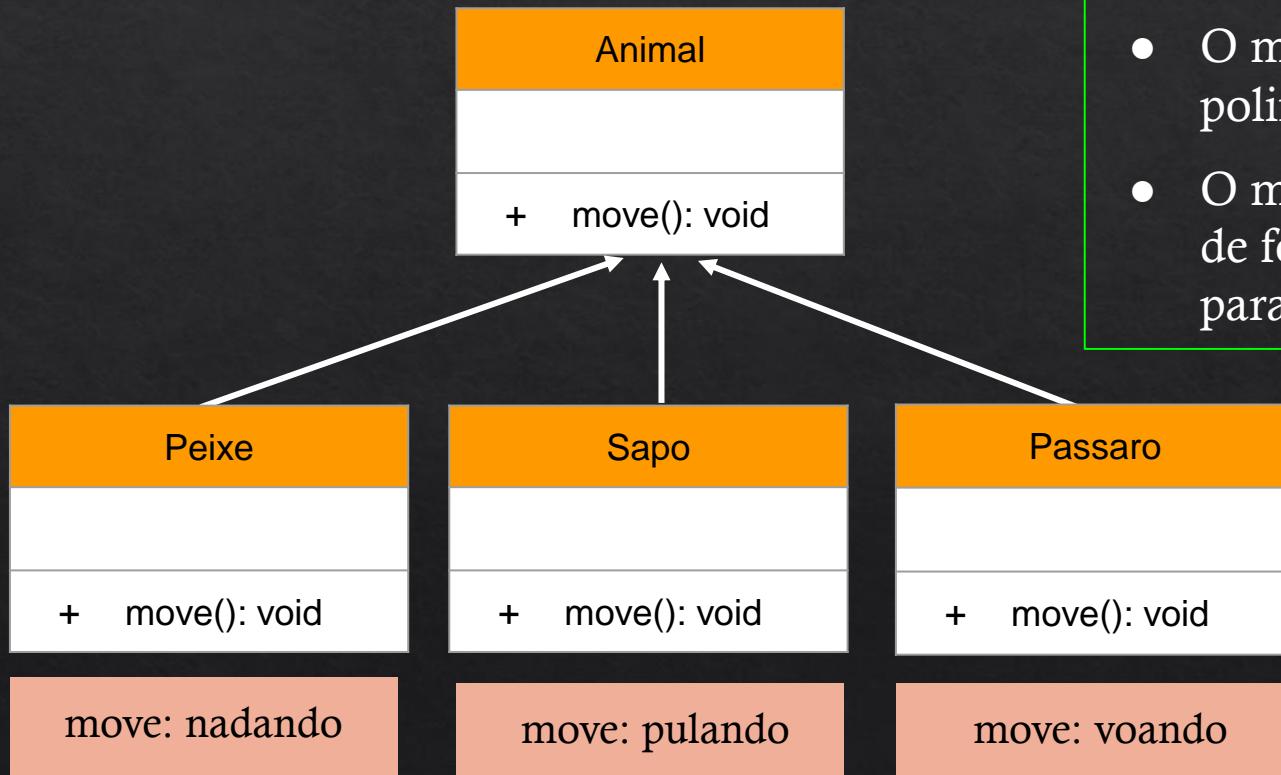
Conceitos de Polimorfismo

- Processa objetos de classes que fazem parte da mesma hierarquia, como se fossem todos objetos da superclasse
- Permite *programar no geral* em vez de *programar no específico*

Método Polimórfico

- Cada objeto sabe fazer a coisa certa em resposta à mesma chamada de método:
 - Diferentes ações ocorrem, dependendo do tipo de objeto!
 - Os **métodos são polimórficos!**

Exemplo - Hierarquia Animal



main:

- O método `move()` é polimórfico!
- O método responde de forma adequada para cada animal

Exemplo - Hierarquia Animal

```
class Animal {  
    public void move() {  
        System.out.println( "Animal anda" );  
    }  
  
}  
  
class Peixe extends Animal {  
    @Override  
    public void move() {  
        System.out.println( "Nada" );  
    }  
  
}  
  
class Sapo extends Animal {  
    @Override  
    public void move() {  
        System.out.println( "Pula" );  
    }  
  
}  
  
class Passaro extends Animal {  
    @Override  
    public void move() {  
        System.out.println( "Voa" );  
    }  
}
```

```
import java.util.ArrayList;  
  
public class MainAnimalPoli {  
    public static void main( String[] args ) {  
        ArrayList<Animal> animais = new ArrayList<Animal>();  
        animais.add( new Peixe() );  
        animais.add( new Sapo() );  
        animais.add( new Passaro() );  
  
        animais.get(0).move();  
        animais.get(1).move();  
        animais.get(2).move();  
    }  
}
```

@Override é uma anotação - serve para indicar que o método da subclasse está sendo sobrescrito / reescrito.

Vantagens do Polimorfismo

- Escalabilidade e Flexibilidade:
 - Com o polimorfismo podemos projetar e implementar sistemas que são facilmente extensíveis.
 - Novas classes podem ser adicionadas com pouco ou nenhuma alteração no programa
- Clareza e manutenção do código com menos linhas

Sobrecarga vs. Sobrescrita de métodos

□ Sobrecarga (*overload*)

- Dentro da mesma classe
- Mesmo nome, porém assinaturas diferentes

□ Sobrescrita (*override*)

- Os método é sobreescrito na subclasse quando já existe na superclasse
- Mesma assinatura

Classes Abstratas

Classes Abstratas

- A classe abstrata é sempre uma **superclasse** que não possui instâncias: **não pode ser instanciada**.
- Ela define um **modelo genérico** para determinada funcionalidade e geralmente fornece uma implementação incompleta dessa funcionalidade
- Cada uma das subclasses da classe abstrata completa a funcionalidade da classe abstrata, adicionando um comportamento específico

Classes Abstratas

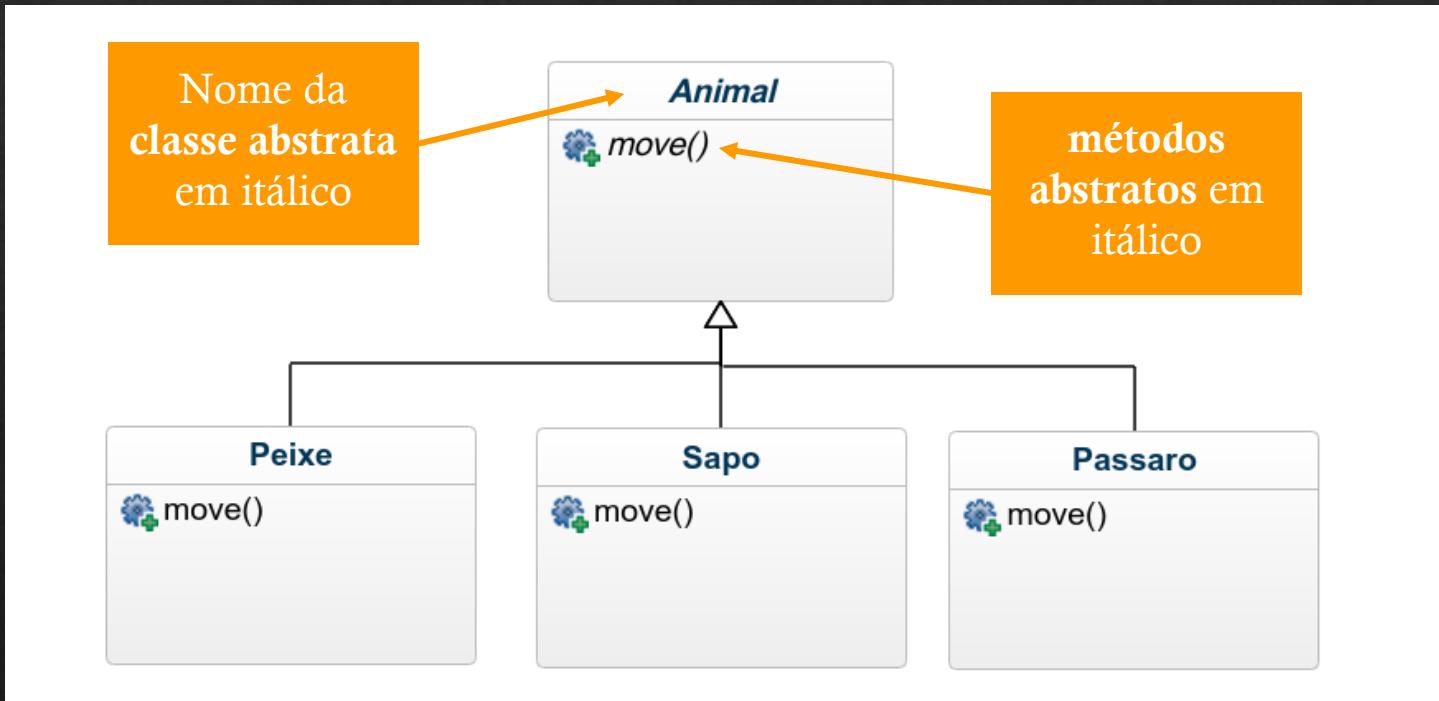
- No Java precisamos especificar explicitamente que a **classe é abstrata**, através da palavra-chave ***abstract***
- Classes abstratas normalmente contém um ou mais **métodos abstratos**, que também são especificados por ***abstract***
 - Métodos abstratos não têm implementação!
 - Definir um método como abstrato é uma maneira de forçar a sua implementação nas subclasses

Classes Abstratas

- No nosso exemplo, a classe *Animal* ficaria:

```
1 ▼ abstract class Animal {  
2     public abstract void move();  
3 }  
4  
5 ▼ class Peixe extends Animal {  
6     @Override  
7     public void move() {  
8         System.out.println( "Nada" );  
9     }  
10 }  
11  
12 ▼ class Sapo extends Animal {  
13     @Override  
14     public void move() {  
15         System.out.println( "Pula" );  
16     }  
17 }  
18  
19 ▼ class Passaro extends Animal {  
20     @Override  
21     public void move() {  
22         System.out.println( "Voa" );  
23     }  
24 }
```

UML - Diagrama de Classes Classe Abstrata



Exemplo completo: Polimorfismo com Classe Abstrata

```
1 abstract class OperacaoMatematica {  
2     public abstract double calcular(double x, double y);  
3 }  
4  
5 class Soma extends OperacaoMatematica {  
6     public double calcular(double x, double y) {  
7         return x + y;  
8     }  
9 }  
10  
11 class Subtracao extends OperacaoMatematica {  
12     public double calcular(double x, double y) {  
13         return x - y;  
14     }  
15 }
```

```
1 import java.util.Scanner;  
2  
3 class Contas {  
4     ...  
5     public static void main(String args[]) {  
6         OperacaoMatematica operacao;  
7  
8         Scanner entrada = new Scanner(System.in);  
9         String op = entrada.nextLine();  
10  
11         if (op.equals("soma"))  
12             operacao = new Soma();  
13         else  
14             operacao = new Subtracao();  
15  
16         System.out.println("Resultado: " + operacao.calcular(5, 4));  
17     }  
18 }
```

Upcasting

e

Downcasting

Upcasting / Downcasting

- São feitos por meio do Polimorfismo
- Tem relação direta com Herança e Hierarquia de Classes

Upcasting

- Com o *upcasting*, podemos nos referir a um objeto da subclasse como sendo da superclasse
- *Upcasting* não precisa ser feito de forma **explícita!**

Upcasting

- Já usamos:

Classe Base / Superclasse: *Funcionario*

Classe Derivada / Subclasse: *Gerente*

Funcionario g = new Gerente()



Upcasting

- É como uma *promoção* do *Gerente* para o tipo *Funcionário*
- O *upcasting* de forma explícita seria:

```
Funcionario g = (Funcionario)new Gerente()
```



Downcasting

- É o processo **contrário** ao *upcasting*!
- A ideia aqui é converter a referência da superclasse para uma referência de uma subclasse
- Sempre precisa ser explícito!

Downcasting

Classe Base / Superclasse: *Funcionario*

Classe Derivada / Subclasse: *Gerente*

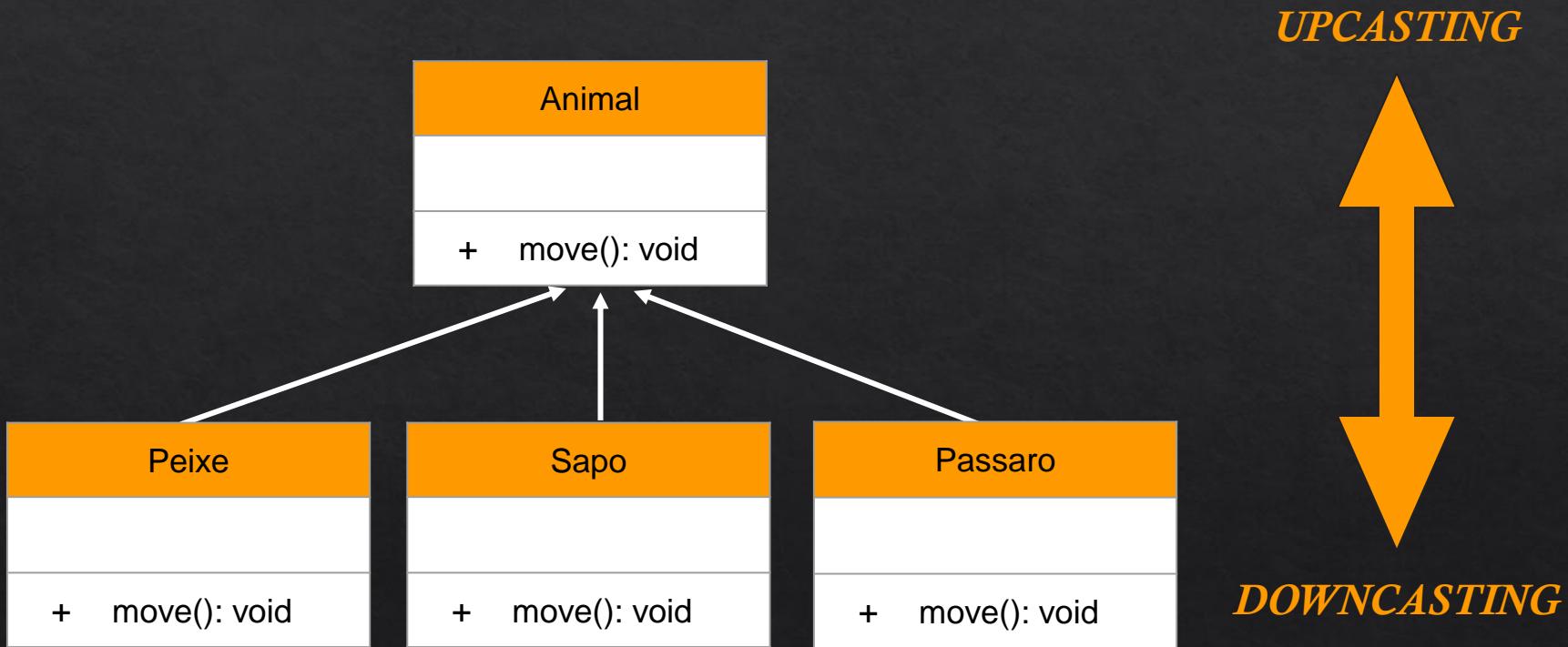
```
Funcionario g = new Gerente()  
Gerente g2 = (Gerente)g
```

Fucionário

Gerente



Exemplo - Hierarquia Animal



Mas, por que usar *Downcasting*?

Exemplo - Hierarquia Animal

```
class Animal {  
    public void move() {  
        System.out.println("Animal anda");  
    }  
  
import java.util.ArrayList;  
  
class Peixe extends Animal {  
    public void move() {  
        System.out.println("Nada");  
    }  
  
    public void fala(){  
        System.out.println("Peixe fala");  
    }  
  
}  
  
class Sapo extends Animal {  
    public void move() {  
        System.out.println("Pula");  
    }  
  
}  
  
class Passaro extends Animal {  
    public void move() {  
        System.out.println("Voa");  
    }  
}
```

```
class MainAnimalPoli {  
    public static void main(String[] args) {  
        ArrayList<Animal> animais = new ArrayList<Animal>();  
  
        animais.add(new Peixe());  
        animais.add(new Sapo());  
        animais.add(new Passaro());  
  
        animais.get(0).move();  
        animais.get(1).move();  
        animais.get(2).move();  
        animais.get(0).fala();  
    }  
}
```

não funciona!

```
MainAnimalPoli.java:37: error: cannot find symbol  
        animais.get(0).fala();  
                           ^  
  symbol:   method fala()  
  location: class Animal  
1 error
```

Exemplo - Hierarquia Animal

```
class Animal {  
    public void move(){  
        System.out.println("Animal anda");  
    }  
  
import java.util.ArrayList;  
  
class Peixe extends Animal {  
    public void move(){  
        System.out.println("Nada");  
    }  
  
    public void fala(){  
        System.out.println("Peixe fala");  
    }  
  
}  
  
class Sapo extends Animal {  
    public void move(){  
        System.out.println("Pula");  
    }  
  
}  
  
class Passaro extends Animal {  
    public void move(){  
        System.out.println("Voa");  
    }  
}
```

```
class MainAnimalPoli {  
    public static void main( String[] args ) {  
        ArrayList<Animal> animais = new ArrayList<Animal>();  
  
        animais.add( new Peixe() );  
        animais.add( new Sapo() );  
        animais.add( new Passaro() );  
  
        animais.get(0).move();  
        animais.get(1).move();  
        animais.get(2).move();  
        ((Peixe)animais.get(0)).fala();  
    }  
}
```

Nada
Pula
Voa
Peixe fala

Funciona!!!

Exercício 2 - Implementar o polimorfismo na hierarquia de classes Formas

□ Classe Formas (*abstrata*)

- Atributo
 - string tipo
- Métodos
 - double perimetro()
 - void print()

□ Classe Círculo

- Atributo
 - double raio
- Métodos
 - double area()
 - double perimetro()
 - void print()

□ Classe Retangulo

- Atributo
 - double comprimento, largura
- Métodos
 - double perimetro()
 - void print()

Interfaces

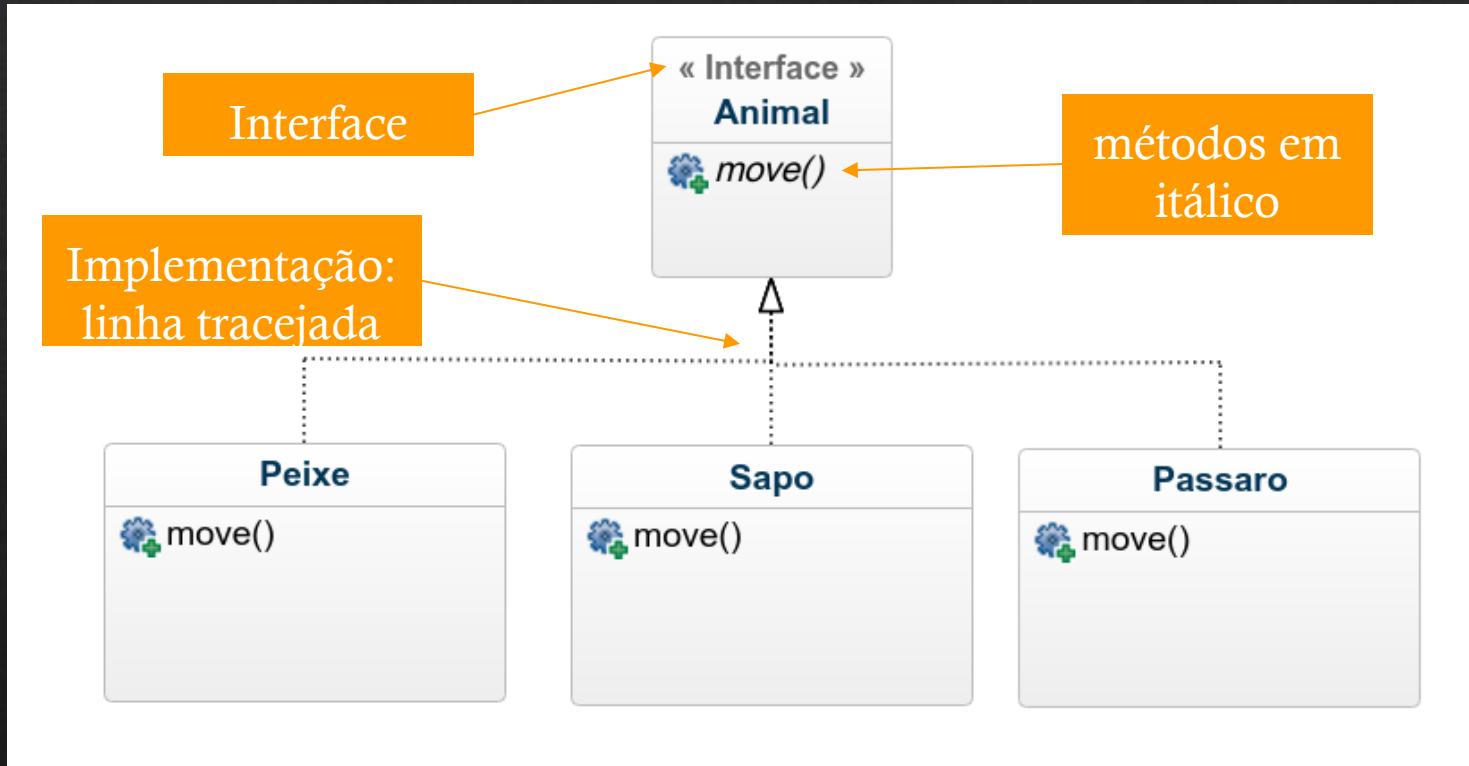
Interfaces

- Interface em Java é o mecanismo pelo qual o programador pode definir um conjunto de operações sem se preocupar com a sua implementação
- A interface indica um modelo de comportamento para outras classes
- Uma interface define um conjunto de métodos que será implementado por uma classe
- As interfaces são semelhantes às classes abstratas com métodos abstratos
- A sintaxe da interface é semelhante a classe abstrata, mas utiliza a palavra-chave **interface** no lugar de **abstract class**.

Interfaces

- A classe que implementa a interface utiliza a palavra-chave *implements* no lugar de *extends*
- A interface **obriga** um determinado grupo de classes a ter métodos ou propriedades em comum
- Pode-se dizer, a grosso modo, que uma interface é um contrato que quando assumido por uma classe deve ser **implementado**

UML - Diagrama de Classes Interface



Interfaces

- No nosso exemplo, Animal poderia ser uma interface

```
1 interface Animal {  
2     public void move();  
3 }  
4  
5 class Peixe implements Animal {  
6     public void move() {  
7         System.out.println( "Nada" );  
8     }  
9 }  
10  
11 class Sapo implements Animal {  
12     public void move() {  
13         System.out.println( "Pula" );  
14     }  
15 }  
16  
17 class Passaro implements Animal {  
18     public void move() {  
19         System.out.println( "Voa" );  
20     }  
21 }
```

```
1 public class MainAnimalPoli {  
2     public static void main( String[] args ) {  
3         Animal peixe = new Peixe();  
4         Animal sapo = new Sapo();  
5         Animal passaro = new Passaro();  
6  
7         peixe.move();  
8         sapo.move();  
9         passaro.move();  
10    }  
11 }
```

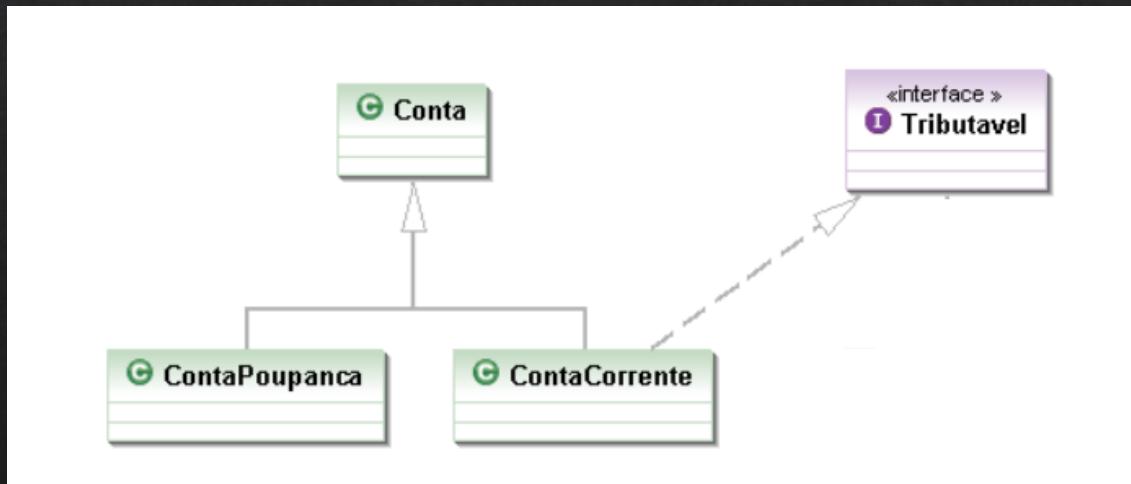
Combinando Interface com Herança

- Uma classe pode herdar comportamentos de outra classe enquanto também implementa uma interface!

```
public class NomeDaClasse extends Superclasse implements Interface{  
    // corpo da classe  
}
```

Combinando Interface com Herança

- Exemplo: Em um banco precisamos tributar alguns bens e outros não: ContaPoupanca não é tributável, já ContaCorrente precisa pagar 1% do saldo em tributos. Para isso, vamos criar a interface *Tributavel*



Combinando Interface com Herança

- Exemplo:

"todos que quiserem ser tributável precisam saber retornar o valor do imposto, devolvendo um double"

```
public interface Tributavel {  
    public double getValorImposto();  
}
```

```
public class ContaCorrente extends Conta implements Tributavel {  
    public double getValorImposto() {  
        return this.getSaldo() * 0.01;  
    }  
}
```

```
public class Conta {  
    private int numero;  
    private String dono;  
    private double saldo;  
    private double limite;  
  
    public double getSaldo(){  
        return saldo;  
    }  
}
```

Por que usar interface?

- Interfaces são muito utilizadas como uma **forma de obrigar** o programador a seguir o padrão do projeto
- O programador é obrigado a implementar os métodos da interface em sua classe, sempre seguindo o padrão
- Interfaces têm a vantagem de não acoplar as classes; ao contrário da herança que traz muito acoplamento (o que pode resultar em quebra do encapsulamento)

Diferenças entre Interface e Classe Abstrata

Métodos

- **Classe abstrata:** podem existir métodos abstratos e não abstratos, públicos, protegidos e privados.
- **Interface:** os métodos são implicitamente abstratos e somente públicos.

Variáveis

- **Classe abstrata:** podem ser definidas variáveis de instância e variáveis de leitura (*final*).
- **Interface:** as variáveis são sempre *public static final*

Quando usar interfaces ou classes abstratas?

Classes Abstratas:

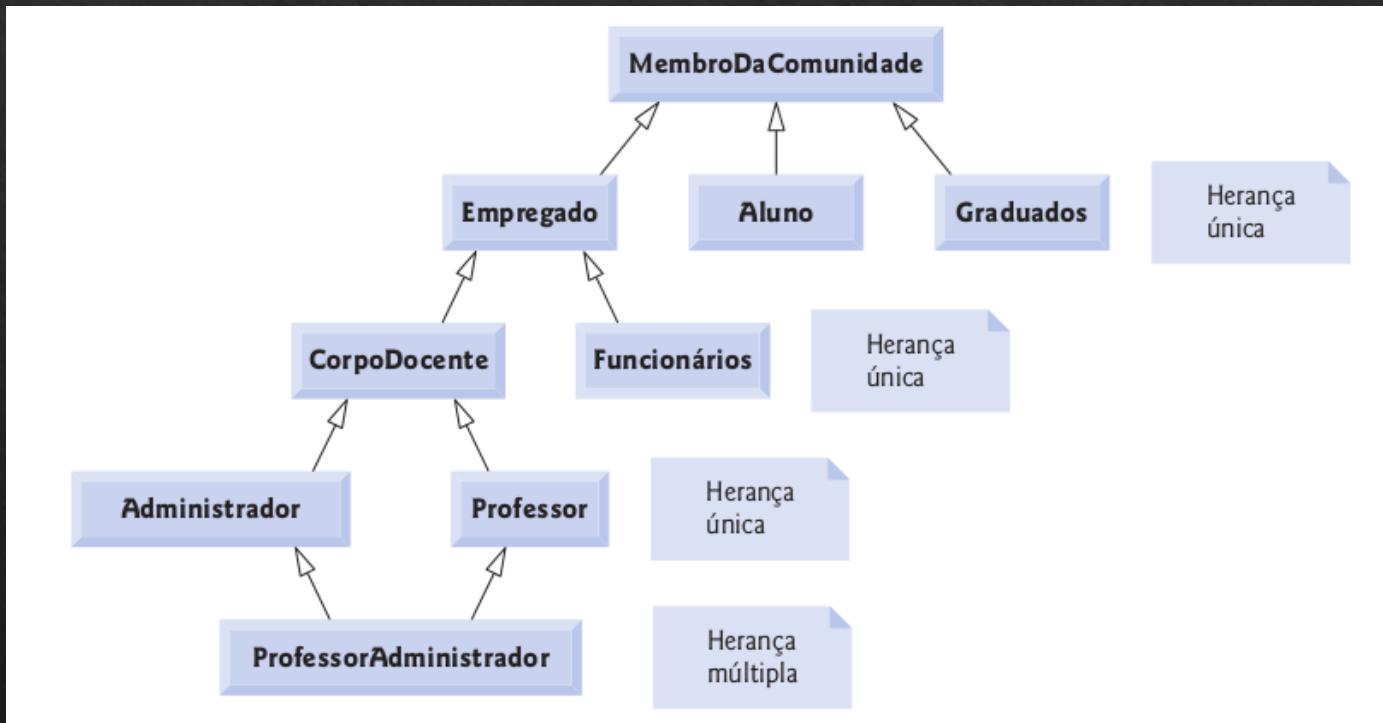
- Você deseja compartilhar código entre várias classes **estreitamente relacionadas**
- Você espera que as classes que estendem sua classe abstrata tenham muitos **métodos ou campos comuns**, ou exigem modificadores de acesso que não sejam públicos (como protegidos e privados)
- Você deseja declarar **campos não estáticos ou não finais**. Isso permite que você defina métodos que podem acessar e modificar o estado do objeto ao qual eles pertencem

Quando usar interfaces ou classes abstratas?

Interfaces:

- Você espera que classes não relacionadas implementem sua interface
- Você deseja **especificar o comportamento** de um determinado tipo de dados, mas não se preocupa com quem implementa seu comportamento
- Você deseja aproveitar a **herança múltipla?!**

Herança Múltipla



Herança Múltipla - Problema

- A herança múltipla pode ser complexa e propensa a erros;
- Pode causar **ambiguidade!**
- Membros das superclasses podem ter o mesmo nome!



- Tanto a classe *Administrador* quanto a *Professor* podem ter o método *getNome()*, por exemplo.
- A classe *ProfessorAdministrador* vai herdar qual dos dois métodos *getNome()* ?!

Interface

Herança Múltipla no Java?!

```
1 interface Andar
2 {
3     default void andar(){
4         System.out.println("Andando!!!");
5     }
6 }
7
8 interface Engatinhar
9 {
10    default void engatinhar(){
11        System.out.println("Engatinhando!!!");
12    }
13 }
14
15 class Animal implements Andar, Engatinhar
16 {
17     public static void main(String[] args)
18     {
19         Animal self = new Animal();
20
21         self.andar();
22         self.engatinhar();
23     }
24 }
```

- A partir do Java 8, as interfaces também podem ter comportamentos
- Isto é possível graças a palavra-chave *default*, utilizada no método
- Então, se uma classe implementar duas interfaces e ambas definerem métodos padrão (*default*), essa nova classe está, essencialmente, herdando comportamentos

Mas, com interface, os problemas da herança múltipla continuam existindo?

```
1 interface Andar
2 {
3     default void mover(){
4         System.out.println("Andando!!!");
5     }
6 }
7
8 interface Engatinhar
9 {
10    default void mover(){
11        System.out.println("Engatinhando!!!");
12    }
13 }
14
15 class Animal implements Andar, Engatinhar
16 {
17     public static void main(String[] args)
18     {
19         Animal self = new Animal();
20
21         self.mover();
22     }
23 }
```

```
multiple_inher_problem.java:15: error: class Animal inherits unrelated defaults
for mover() from types Andar and Engatinhar
class Animal implements Andar, Engatinhar
 ^
1 error
```

Interface

Herança Múltipla no Java?!

Para resolver o conflito, a classe deve decidir qual método *mover()* deseja invocar e então chamar usando a *referência da interface* e a palavra-chave *super*.

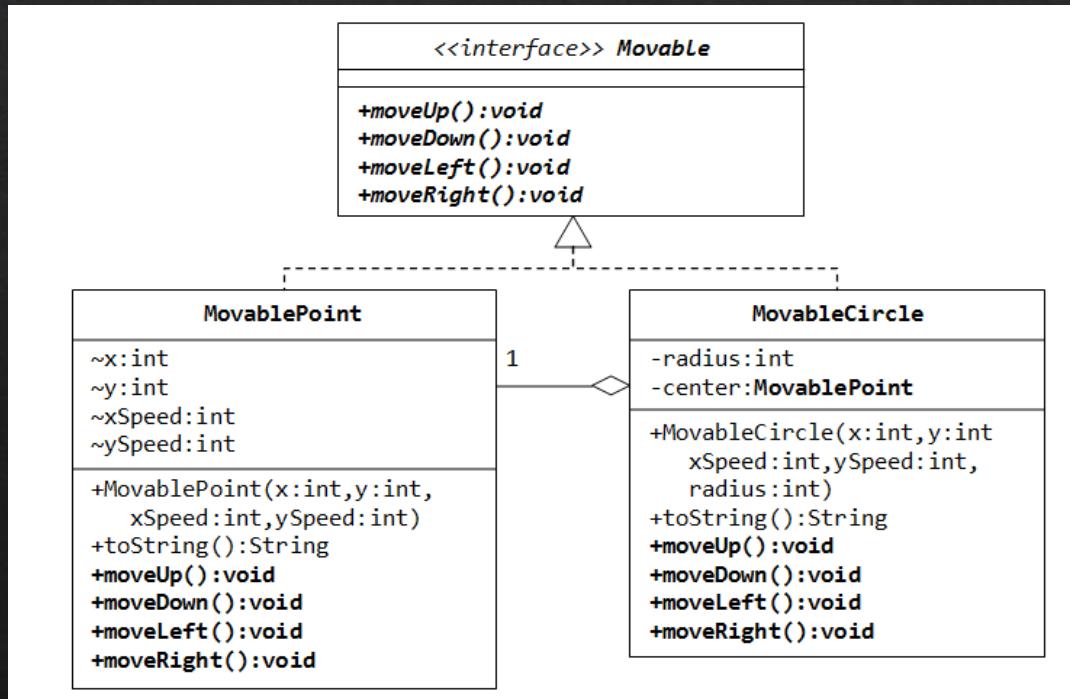
```
1 interface Andar
2 {
3     default void mover(){
4         System.out.println("Andando!!!");
5     }
6 }
7
8 interface Engatinhar
9 {
10    default void mover(){
11        System.out.println("Engatinhando!!!");
12    }
13 }
14
15 class Animal implements Andar, Engatinhar
16 {
17     public void mover(){
18         Andar.super.mover();
19     }
20 }
21
22
23 class Teste{
24     public static void main(String[] args)
25     {
26         Animal a = new Animal();
27         a.mover();
28     }
29 }
```

Exercício 1 - Sobrecarga de Método

Crie uma classe chamada *Area* para imprimir a área de um quadrado e de um retângulo. A classe deve ter, portanto, dois métodos com o mesmo nome, mas diferentes números de parâmetros. O método para imprimir a área do retângulo tem dois parâmetros que são comprimento e largura, respectivamente, enquanto o outro método para imprimir a área do quadrado tem somente um parâmetro que é o lado do quadrado.

Exercício 2

Implemente o Diagrama de Classes apresentado abaixo.



Modificador de acesso ~ (til): Os atributos ou métodos podem ser acessados por qualquer objeto dentro do pacote

No Java, basta não colocar nada: package-private

Exercício 3 - Matrix (Saraiva Jr., O. - Cap. 9)

“Neo, a Matrix é um grande sistema orientado a objetos, em que as pessoas são objetos de classes que herdam da classe abstrata Agente.”

Dada a definição acima de *Matrix*, implemente as classes *Empresario*, *Professor* e *Advogado* como subclasses da classe *Agente*:

- A classe *Agente* tem os seguintes atributos: String nome; boolean modo_agente; String profissao; o método abstrato void apresentacao() e o método modo_agente_on();
- A classe *Empresario* deve ter os atributos: String nome; boolean modo_agente; String profissao, String empresa; e o método void apresentacao();
- A classe *Professor* deve ter os atributos: String nome; boolean modo_agente; String profissao, String escola; e o método void apresentacao();
- A classe *Advogado* deve ter os atributos: String nome; boolean modo_agente; String profissao, String OAB; e o método void apresentacao();

Exercício 3 - *Matrix* (Saraiva Jr., O. - Cap. 9)

No método *main()* a explicação de *Matrix* deve ser implementada.

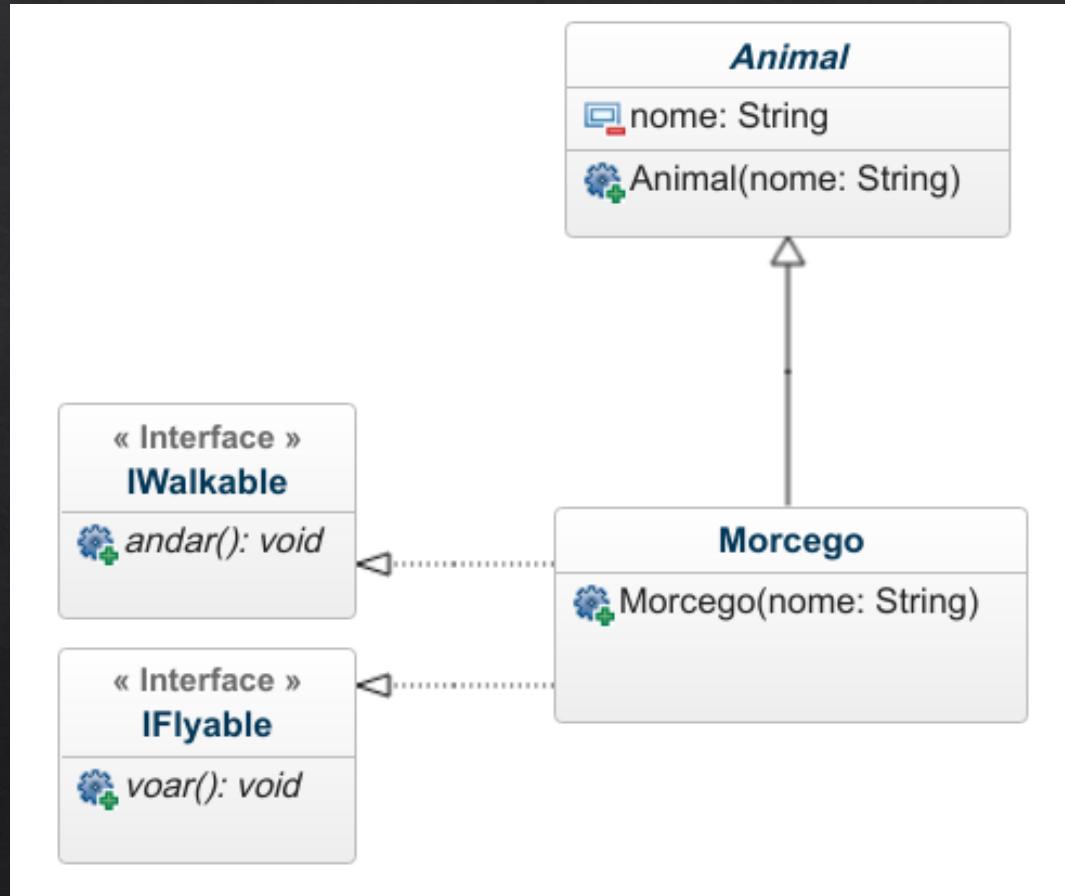
Um ArrayList de *n* pessoas do tipo Agente deve ser definido. Então, com um loop infinito, faça um menu para criar pessoas com diferentes profissões dentro deste ArrayList.

Coloque uma opção no menu para todas as pessoas se apresentarem.

Escolha algumas pessoas para invocar o método *modo_agente_on()* e transformá-las em Agentes. Cuidado: os objetos não podem ser instanciados como Agentes (classe abstrata), mas inclua um print no método *apresentacao()* que leia o atributo *modo_agente* e imprima “AGENTE SMITH”, para *modo_agente = true*.

Exercício 4

Implemente o Diagrama de Classes ao lado. Crie também uma classe para testar o funcionamento de objetos da classe Morcego.



Obrigada pela sua
participação, nos vemos na
próxima aula! :)