PROGRAMAÇÃO ORIENTADA À OBJECTOS

INTERFACES

INTRODUÇÃO

Até aqui discutimos os conceitos relacionados com herança e as vantagens que ela nos dá no mecanismo de polimorfismo. Abordamos também as vantagens do uso de classes e métodos abstractos. E, quais são as vantagens do polimorfismo e da abstracção? Reveja! Neste artigo iremos introduzir *interfaces.*

Voltemos a nossa atenção ao exemplo da ficha sobre herança. Depois que introduzimos abstracção, ficamos com uma classe abstracta (a classe *Animal*) que assinou o métodos *calculaIdade.* As nossas classes *Mamífero, Insecto e Peixe* estenderam a classe *Animal* e viram se obrigadas a implementar o método *calculaIdade* devido ao facto de este ter sido definido como abstracto na super classe (*Animal*). Agora, vamos supor que quiséssemos introduzir mais uma classe no nosso sistema, a classe Ave. Vamos supor ainda que quiséssemos fazer os mamíferos e as aves "caminhar", ou melhor, fazer com que todos os objectos das classes de animais com pernas ou patas caminhassem. Para isso teríamos de introduzir um método, *caminhar.* Vamos supor que o "caminhar" seja incrementar o valor de um atributo o qual chamaremos de *quilómetros percorridos*. Mas em qual das classes? Será que colocaríamos este método na super classe *Animal?* Não! Afinal nem todos os animais caminham. O "caminhar" só faz sentido para animais com patas. Logo este método teria de ser colocado somente na classe *Mamífero* e *Ave*. Nossas classes ficariam assim:

```
public class Ave extends Animal{
    int kmPercorridos;

    //Atributos e metodos do ave

    public double caminhar(){
        kmPercorridos += 1;
        return kmPercorridos;
    }
}
```

```
public class Mamifero extends Animal{
    int kmPercorridos;

    //Atributos e metodos do mamifero

    public double caminhar(){
        kmPercorridos += 2;
        return kmPercorridos;
    }
}
```

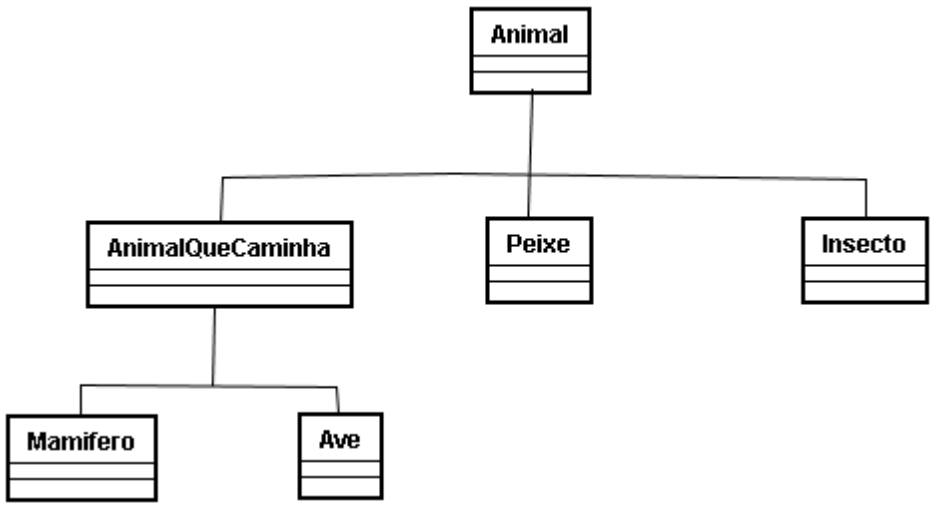
Desta forma não será possível usar o polimorfismo!

Vamos supor que quiséssemos criar agora um método (algures em um programa *ControleDeDistanciasPercorridas*) para fazer um animal caminhar. Repare na situação apresentada a seguir.

```
public class ControleDeDistanciasPercorridas {
    public static void fazerCaminhar(Mamifero m){
        System.out.println("Ja caminhou: " + m.caminhar());
    }
    ...
}
```

Para a ave seria necessário um método idêntico que receberia como parâmetro uma Ave. Logo teríamos de criar dois métodos iguais! Mas este problema já havia sido resolvido usando o polimorfismo! E o que teríamos de fazer para conseguir polimorfismo? Definir uma super classe para Mamíferos e Aves. Essa super classe juntaria as características comuns dos animais que caminham.

E a estrutura das nossas classes ficaria assim:



Agora nosso *fazerCaminhar* ficaria assim:

```
public void fazerCaminhar(AnimalQueCaminha a){
    System.out.println("Ja caminhou: " + a.caminhar());
}
```

Vamos supor que agora quiséssemos introduzir mais uma entidade no nosso sistema que tivesse a habilidade de caminhar, digamos, um Robô. É sabido que o robô caminha, como faríamos para fazer o robô "caminhar", isto é, fazer o uso do método *fazerCaminhar* do nosso sistema? A solução seria fazer o robô estender a classe *AnimalQueCaminha* e desta forma nos beneficiaríamos do uso de polimorfismo para as três entidades (*Mamífero*, *Ave* e agora, *Robô*). A nossa classe *Robô* ficaria assim:

```
public class Robo extends AnimalQueCaminha {
        //Codigo da classe robo
}
```

Mas algo não cheira bem aqui. Será que um robô é um animal? Não! Logo, o relacionamento que estabelecemos entre o *Robô* e o *AnimalQueCaminha* é um absurdo! Nosso objectivo era usar de polimorfismo e evitar a reescrita do método *fazerCaminhar* do nosso sistema. Este tipo de relacionamento poderá nos trazer problemas sérios no futuro, logo é melhor nos abstermos de fazer este absurdo!

Mas como resolver o problema? Afinal queremos evitar a repetição do método *fazerCaminhar!* Isto é, usar o mesmo método tanto para um mamífero, como para uma ave, bem como para um robô.

Isso só seria possível se pudéssemos referenciar as três entidades em causa (Mamífero, Ave e Robô) da mesma forma. O que estas entidades têm de comum é o método *caminhar* que é usado pelo método *fazerCaminhar.* E se pudéssemos passar como parâmetro para o método *fazerCaminhar* uma entidade capaz de "caminhar", independentemente de ser *Ave*, *Mamífero* ou *Robô*? Afinal o caminhar é a característica que importa para fazer uma entidade caminhar!

Existe em java uma forma de garantir que um conjunto de entidades (classes) tenha (obrigatoriamente) certas habilidades em comum sem que estas (entidades) sejam subclasses de uma classe comum. Esse mecanismo chama-se interface

DEFINIÇÃO

Uma interface é uma classe que define um conjunto de habilidades que devem ser implementadas por todas as classes que precisam ser tratadas duma determinada forma [definida pela interface]. Usando o mecanismo de interface é possível referenciar objectos de várias classes da mesma forma sem que estas sejam descendentes da mesma super classe.

Como este mecanismo resolve o problema levantado? No nosso problema estávamos interessados em referenciar o *Mamífero*, a *Ave* e o *Robô* da mesma forma para poder usar o método comum *fazerCaminhar.* Usando Herança fracassamos; Mas usando interfaces seremos bem sucedidos pois conseguiremos os nossos intentos: *referenciar as três classes da mesma forma.*

Para isso vamos criar uma interface que deverá ser implementada por todas as classes com a habilidade "*caminhar".* Vamos chamar a interface de *EntidadeQueCaminha.* O código desta interface fica assim:

```
public interface EntidadeQueCaminha {
    public double caminhar();
}
```

Note que a interface define apenas as assinaturas dos métodos e nunca a implementação destes (tal como acontece com métodos abstractos). A implementação dos métodos deverá ser feita *obrigatoriamente* nas classes que a implementam.

Nossas classes agora terão a seguinte aparência.

```
public class Mamifero extends Animal implements EntidadeQueCaminha{
    int kmPercorridos;

    public double caminhar(){
        kmPercorridos += 1;
        return kmPercorridos;
    }

    //Outros atributos e metodos da classe Mamifero
}
```

```
public class Robo implements EntidadeQueCaminha {
    int kmPercorridos;

    public double caminhar(){
        kmPercorridos += 1;
        return kmPercorridos;
    }

    //Outros atributos e metodos do Robo
}
```

```
public class Ave extends Animal implements EntidadeQueCaminha{
    int kmPercorridos;
    public double caminhar(){
        kmPercorridos += 1;
        return kmPercorridos;
    }

    //Outros atributos e metodos da Ave
}
```

Já podemos nos referir a cada uma destas entidades como *EntidadeQueCaminha*. Por exemplo, podemos fazer, sem nenhum problema, as seguintes referências.

```
EntidadeQueCaminha ave = new Ave();
EntidadeQueCaminha robo = new Robo();
```

E podemos reescrever o nosso método *fazerCaminhar* do nosso sistema da seguinte maneira:

```
public class ControleDeDistanciasPercorridas {
    public static void fazerCaminhar(EntidadeQueCaminha e){
        System.out.println("Ja caminhou: " + e.caminhar());
    }
}
```

E algures num *main* podemos fazer o seguinte:

```
public class ControleDeDistanciasPercorridas {
    public static void fazerCaminhar(EntidadeQueCaminha e){
        System.out.println("Ja caminhou: " + e.caminhar());
    }

    public static void main(String[] args){
        EntidadeQueCaminha ave = new Ave();
        EntidadeQueCaminha robo = new Robo();

        fazerCaminhar(ave);
        fazerCaminhar(robo);
    }
}
```

Note que o criador do método *fazerCaminhar* não precisa de se preocupar com os detalhes do método caminhar; ele simplesmente precisa saber que efeitos este método produz, e nunca como isso é feito. Uma interface funciona como um controle de um aparelho electrónico. O utilizador do controle usa-o para manejar o aparelho e nunca se preocupa com os detalhes do funcionamento interno deste. Por exemplo, o usuário sabe que o botão "on/off" liga e desliga o aparelho mas não lhe interessa saber como o liga/desliga é feito.

EXERCÍCIOS

1. Altere o exercício 3 da ficha 10 de modo que o método *calculaSaldoReal* seja definido em uma interface que deverá ser implementada pelas classes que antes faziam uso dele.
2. Ilustre o uso de polimorfismo baseando-se na interface criada no exercício anterior bem como nas classes que a implementam.